CAPÍTULO 1

ALGORITMOS Y TÉCNICAS DE PROGRAMACIÓN

1.1 ALGORITMOS.

Un algoritmo es una secuencia ordenada de operaciones bien definidas y efectivas, que al ser ejecutadas, siempre producen un resultado y terminan en un tiempo infinito Veamos que significa esto.

- Secuencia Ordenada: Indica que el próximo paso está definido sin ambigüedad así como el comienzo y el final del algoritmo.
- Operación bien definida: Claramente entendible a la persona o máquina que la ejecuta.
- Operación efectiva: existe un método formal para ejecutarla y obtener un resultado.
- Termina en un tiempo finito: el algoritmo produce su resultado en un tiempo adecuado al problema que resuelve.

1.2 TÉCNICAS DE PROGRAMACIÓN.

- Análisis Descendentes: Divide & Conquer.
- Programación estructurada: Sin Goto
- Programación Modular: Varios módulos.
- Programación Orientada por Objetos.

1.2.1 PROGRAMACIÓN MODULAR.

Consiste en realizar varios programas con funciones específicas, para luego unirlos y obtener el programa deseado. Inicialmente el programa es dividido en elementos con nombres y direcciones separadas llamadas MODULOS, que se integran para satisfacer los requerimientos del problema.

Ventajas:

- Permite que el programa sea manejable intelectualmente: fácil de leer y comprender.
- Es más fácil resolver un problema complejo cuando se parte en trozos manejables.
- El esfuerzo para desarrollar un módulo es menor a medida que hay más módulos esto es en cuanto a su función y aplicación y no en cuanto a las interfaces entre módulos. “Dividirlo indefinidamente no significa menos complejo”.
- Es más fácil concentrarse en un problema independiente de los detalles de bajo nivel.

- Es más fácil el mantenimiento de los programas (agregar nuevos módulos, expandir o modificar los existentes).

En la programación modular dividimos un programa en módulos que llaman a otros módulos y estos a su vez llamen (o invocan) a otros. Un diagrama de módulos se puede observar en la figura 1.

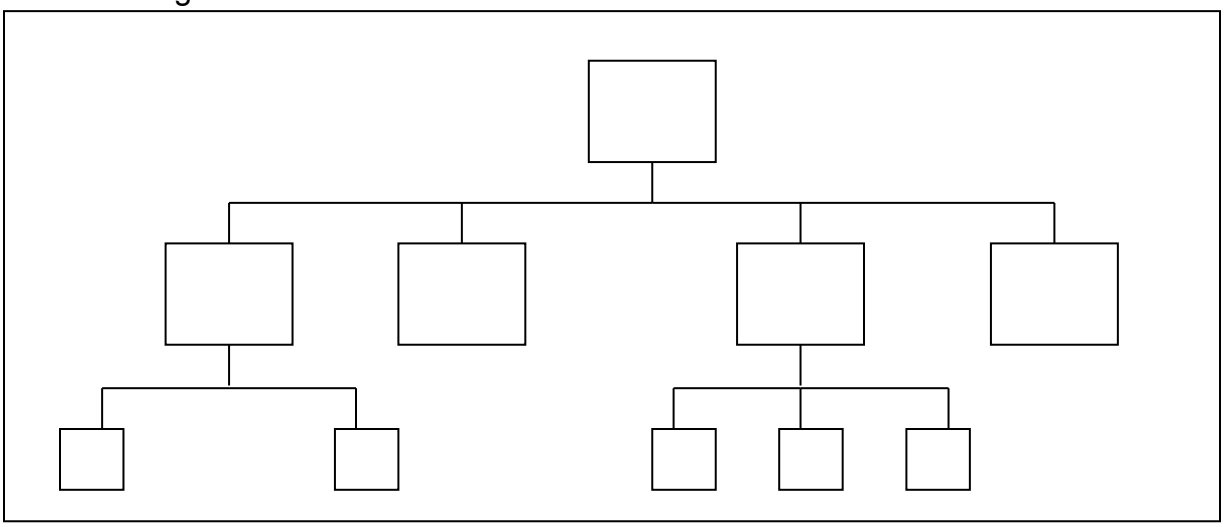

Figura 1.1. Diagrama de módulos

1.2.1.1 Cohesión

El requerimiento más importante de un módulo es que realice una única tarea coherente. De aquí viene una medida conocida como COHESIÓN que es el grado de interrelación de los elementos internos de un módulo.

Otra medida es el ACOPLAMIENTO o grado de dependencia entre dos módulos cada modulo debe ser los mas independiente posible.

Razones para crear módulos coherentes:

- Facilidad de modificación: cambiar el menor código posible.

En caso de necesitar que el programa presente otras características, solo se modifica el módulo necesario en lugar de todo el programa. La sección de código que no es afectada por la nueva especificación no debe ser afectada por el proceso de Modificación. En la figura 1.2 se puede observar el efecto de cambiar j por i en el primer bloque. Si los bloques están acoplados (figura 1.2a) afectaría ambos bloques, mientras que en la figura 1.2b solo afecta el bloque A

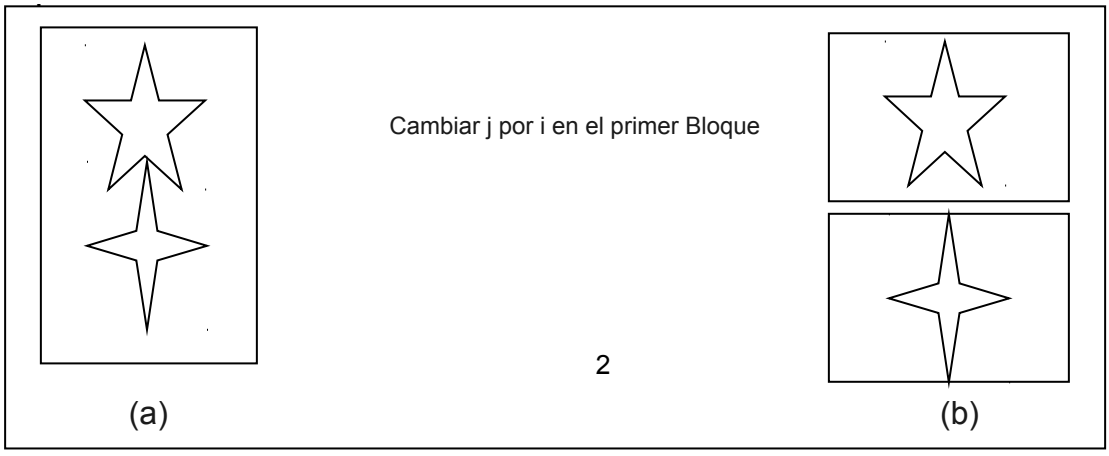

Figura 1.2. Efecto de la cohesión

- Beneficio en la fase de prueba

Permite ir efectuando programas por separado que cumplan funciones específicas probarlos por separado, para luego unirlos y obtener uno con mayor confiabilidad. Esto puede observarse en la figura 1.3 donde vemos un bloque con m caminos en la primera parte y n caminos en la segunda parte. Para probar exhaustivamente este módulo se requiere de mxn pruebas.

En el caso de la figura 1.3b, al tener los bloques separados en dos módulos coherentes, cada módulo se prueba por separado por lo que el total de pruebas necesarias es m+n.

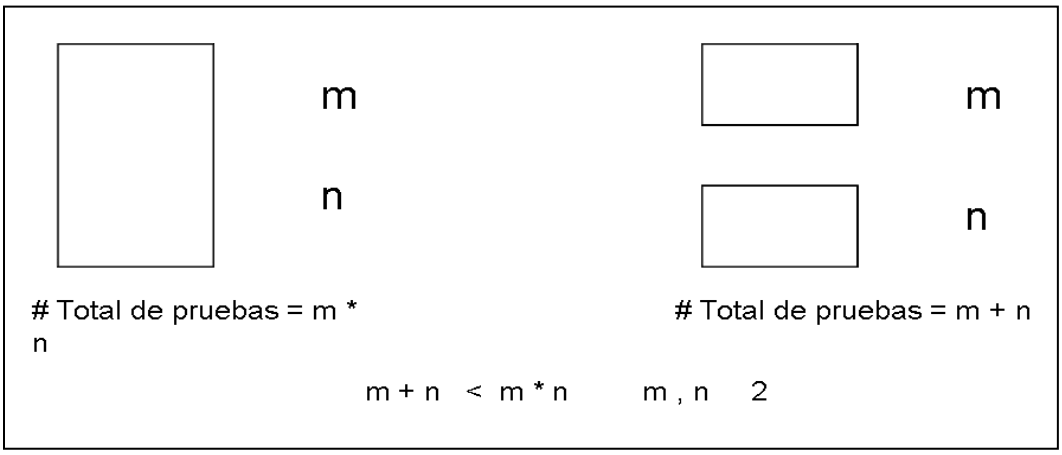

Figura 1.3. Beneficio en la fase de prueba

1.2.1.2 Independencia o desacoplamiento

Cada módulo debe ser tan independiente que al quitarlo del programa y colocar otro que realice la misma función, no se afecta el resto del programa (siempre que las especificaciones de entrada y salida sean idénticas). Ver el ejemplo de la figura 4.

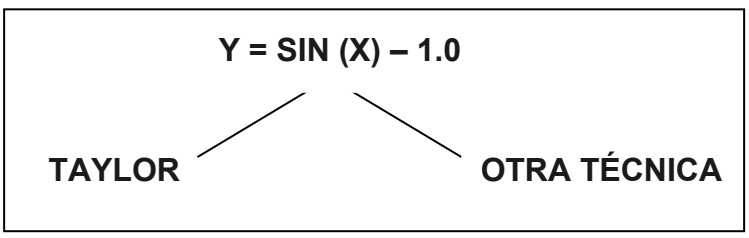

Figura 1.4. Desacoplamiento

Note que al aumentar la cohesión baja el acoplamiento.

1.2.2 ENCAPSULAMIENTO

También conocido como técnica de ocultamiento de información. La idea es que el módulo, que invoca a otro módulo no conozca los detalles internos de implementación del segundo módulo. Es decir, para el primer módulo debe ser indiferente: las estructuras de datos y el código de las operaciones del segundo. Todo módulo debe diseñarse de manera que la información (datos y procedimientos) sea inaccesible por otros módulos que usan tal información. El contenido del módulo es desconocido, es como una caja negra que realiza una tarea sin saber cómo la hace. Los programas que requieren de esa tarea hacen uso de éste invocándolo, dándole los datos necesarios para su funcionamiento. Esto produce: MODULOS INDEPENDIENTES, que se comunican con otros sólo a través de la información necesaria.

VENTAJAS:

- Facilidad de modificación: ya que todo se conoce donde esta localizado.
- Los errores inadvertidos se propagan con menos probabilidad a otras posiciones del programa. Se evita el efecto de ondas

1.3 ALCANCE DE VARIABLES

Se conoce como alcance de una variable a la porción del programa en la cual la definición de un identificador es válida. Consiste en saber que jerarquía poseen las variables del programa, esto se determina dependiendo de donde se encuentren declaradas las variables. En el caso de lenguaje C, la regla de alcance es la siguiente:

"Todo identificador es válido en el bloque donde se declara (módulo o función) y en los bloques internos a menos que haya otro identificador con el mismo nombre dentro del bloque interno".

Cuando haya un conflicto de nombre, el identificador siempre se refiere a la declaración mas reciente. De acuerdo al alcance las variables se pueden clasificar en: variables globales, variables locales, automáticas, estáticas, externas.

- Variables Globales: Son aquellas que afectan a todo el programa, su alcance es global.
- Variables Locales: Son aquellas que únicamente pueden ser observadas dentro del subprograma donde fueron declaradas. Se dice que su alcance es local a dicho programa.
- Automáticas: Son aquellas que desaparecen después de hacer uso de ellas en los programas.
- Estáticas: Mantiene su valor después de la ejecución del subprograma y deben ser procedidas de la palabra static.
- Externas: Variables definidas en un módulo externo, es decir, su definición aparece en otro módulo. Al declararlas en el módulo actual, pueden ser usadas y modificadas en este nuevo contexto. Aumentando así su alcance.

Ejemplo:

```
int a,b,c;
extern x;
main() {
     a=1;
     b=2;
```

```
      c=3;
      P1();
      printf("%d%d%d", a,b,c} ;
      P2();
}

void P2() {
      int b;
      a=4;
      b=5;
      printf("    ",a,b,c);
}
```

En este programa a,b,c de la primera línea son variables globales, pueden ser usadas dentro del main y del procedimiento P2. Excepto b cuyo alcance finaliza con la declaración de la variable b interna al procedimiento P2. Esta variable es de alcance local. La variable es declarada extern, lo cual significa que su definición aparece en otro módulo. La variable b de l procedimiento P2 es estática.

CAPÍTULO 2

MANEJO DE MEMORIA

Podemos imaginarnos la memoria del computador como un gran arreglo formado por casillas o palabras; como se ilustra en la figura 2.1. Cada casilla está constituida por su contenido y su definición física. El contenido de una casilla puede ser datos o instrucciones de programas.

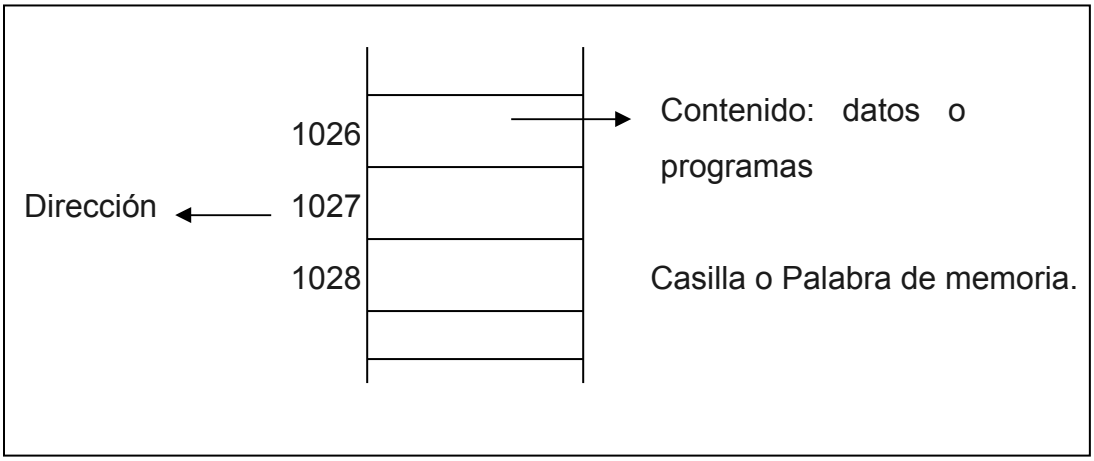

Figura 2.1. Características de memoria

Si la memoria es de 4 Mbytes y cada palabra es de 4 bytes tendríamos 2^{22} palabras de memoria, tenemos los cálculos.

$$4Mb = 4*1024 \text{ Kbytes} = 4* 2^{10} * 1024 \text{ bytes} = 2^2 * 2^{10} * 2^{10} \text{ bytes} = 2^{22} \text{ bytes}.$$

$$2^{22} \text{ palabras de memoria} = \frac{2^{22} \text{ bytes}}{2^2 \text{ bytes}} = 715.776 \text{ palabras}.$$

En cada casilla se almacenan valores de datos o instrucciones de programas (código de un programa traducido a código máquina). El computador toma de la memoria esta información para producir los resultados exigidos por el usuario del computador.

2.1 APUNTADORES

Entre los datos que pueden almacenarse dentro de la memoria están las direcciones a otros datos. A estos se les conoce como apuntadores.

Cuando se almacenan datos en la memoria, a éstos se les asigna un ESPACIO DE MEMORIA según el tamaño dependiendo de la definición del dato.

Por ejemplo (figura 2.2), si tenemos:

```
int  X;        / * X ocupará 2 bytes de memoria.*/
long Y;        / * Y ocupará 4 bytes de memoria.*/
char C;        / * C ocupará 1 bytes de memoria.*/
```

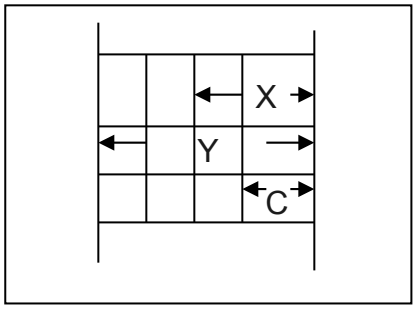

Figura 2.2. Espacio de Memoria

En el caso de los apuntadores, generalmente ocupan una casilla o palabra de memoria y el contenido indica la dirección de otro dato (ver figura 2.3)

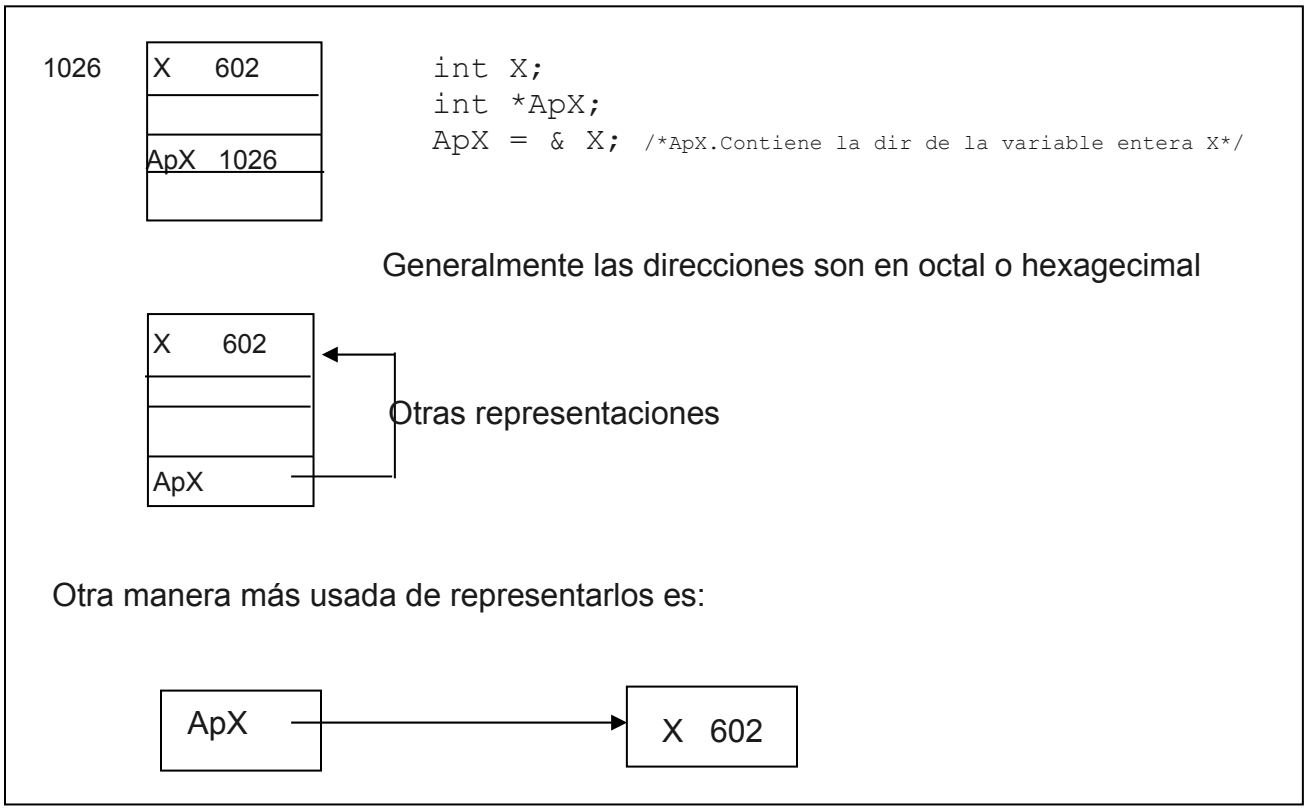

Figura 2.3. Representación de manejo de memoria

2.1.1 OPERADORES

Los apuntadores poseen dos operaciones básicas:

a) Operador de direccionamiento (&):

Operador unario que devuelve la dirección de un objeto.

```
ApX = &X; /* asigna la dirección de X a ApX */
          /* Ahora ApX "apunta a X" */
```

Este operador es solo aplicable a variables y elementos de un arreglo. Vemos ejemplos de su uso.

ILEGALES	LEGALES
&(X+1)	&X
&3	&a[4]

También es ilegal obtener la dirección de una variable **Register.**

b) Operador de indireccionamiento (*)

Operador unario que toma un operando como una dirección y acerca esa dirección para obtener su contenido.

```
int y;
y = *Apx;      /*asigna a y el contenido de Apx*/
```

2.1.2 DECLARACIÓN Y USOS

Para declarar un apuntador se coloca la siguiente instrucción

```
<tipo> *<nombre>;
```

Por ejemplo,

```
int *ApX;
```

indica que el contenido de `ApX` es de tipo `int`, o `*ApX` es equivalente a una variable de tipo entero, por lo tanto puede estar en una expresión donde aparezcan valores enteros. Veamos el ejemplo de usos válidos del apuntador ApX con el operador *.

```
if ((*ApX) + Y == 8)
…

Y = *ApX + 1;

*ApX = 0;

(*ApX) + = 1;    /* Para incrementar, el contenido se
                 necesitan los paréntesis, de lo
                 contrario incrementa la dirección*/

int *ApY;
ApY = ApX;     /* ApY apuntan al mismo contenido */
```

2.1.3 APUNTADORES Y ARGUMENTOS DE FUNCIONES.

Todo argumento de una función en C es "por valor". Al invocar la función se hace una copia del valor del parámetro real (el de la invocación) en el parámetro formal. Esto quiere decir que el argumento conserva su valor después que se regresa de la invocación.

Por ejemplo, veamos el siguiente programa principal que contiene una llamada a una función que intercambia los dos valores a y b

```
main ()
{
   int a,b;
   a=4;
   b=7;
   Swap(a,b);
}
```

En este caso necesito que se modifique los argumentos. Si la función esta definida como:

```
void Swap (int x , int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Al ejecutar el programa con los valores `a=4` y `b=7`, el efecto de `Swap(a,b)` es ninguno aunque dentro de la definición de la función se escribe explícitamente que cambiaron.

La forma de hacer que los valores de `a` y `b` salgan intercambiados es a través de apuntadores, es decir haciendo la siguiente invocación, donde se dice explícitamente que lo que se pasa es la dirección de las variables a y b.

```
Swap(&a,&b);
```

Ahora, lo que no se modifica es la dirección de `a`, y la `b`; pero si puede modificarse su contenido. La definición de la función quedaría:

```
void Swap(int *Apx, int *Apy)
{
   int temp;

   temp = *Apx;
   *Apx= *Apy;
   *Apy = Temp;
}
```

Los apuntadores como argumentos suelen usarse cuando la función devuelve más de un resultado.

2.1.4 APUNTADORES Y ARREGLOS

Los nombres de los arreglos son apuntadores constantes (ver figura 2.4) a la primera casilla del arreglo. Por eso cuando se pasan como parámetro no debe colocarse el operador & para modificar el contenido de sus casillas, basta con pasar el nombre.

Para asignar a un apuntador la dirección de un arreglo, basta con asignarle el nombre, como se observa en la figura 2.4. Lo cual es equivalente a colocarle la dirección a la primera casilla.

Después de esta primera asignación se puede trabajar sobre el arreglo a través del apuntador, con lo que se conoce como aritmética de apuntadores.

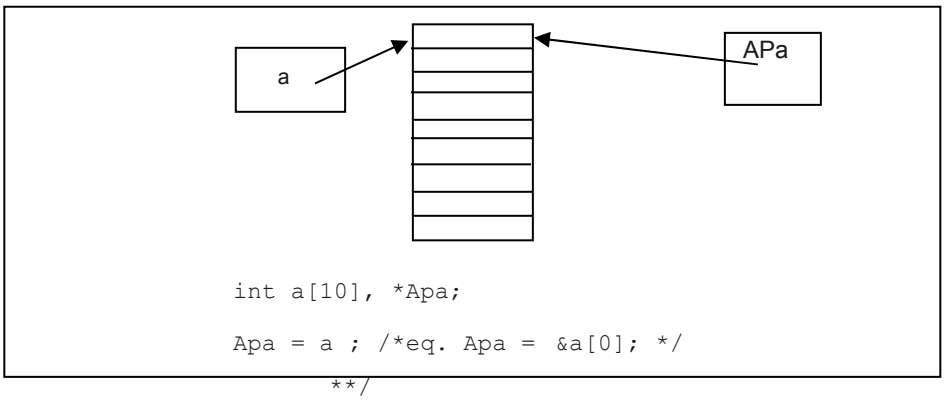

Figura 2.4. Apuntadores

Es decir, podemos escribir instrucciones como

```
Apa++;      /*Para ubicar el apuntador sobre la segunda casilla del arreglo */
Apa--;      /*Para regresarlo a  la primera casilla*/
Apa+=5;     /*Para ubicarlo cinco casilla después*/
```

En algunos casos estas expresiones son útiles, pero si tenemos que elegir entre el manejo cotidiano de los arreglos y el trabajo a través de apuntadores, debemos preferir la sencillez y claridad de la primera alternativa.

2.2 Uso de prototipos

Toda definición debe aparecer antes de ser usada. Esto incluye las definiciones de funciones. Para solventar esto se usan los **prototipos.** Son los encabezados de las funciones que le indican al precompilador que posteriormente encontrará la definición de la función.

Los prototipos se conocen antes del **main** y pueden ir en un archivo header " *.h ".

Ejemplo, En el programa donde aparece el procedimiento `swap` debe haber un prototipo del procedimiento antes del `main` como se muestra a continuación:

```
#include <stdio.h>
void Swap (int *apx; int *apy);
main() {
   int a,b;
   a=4;
   b=7;
   Swap(&a,&b);
}
```

Colocar los prototipos con la descripción de la función y de sus argumentos en un archivo header es un buen estilo de programación.

2.3 Pasaje de Parámetros

Los parámetros que van en la definición de la función o en un prototipo son los **parámetros formales,** estos son como variables locales a la función que sirven de comunicación con el exterior.

Los parámetros reales son los que se colocan en la invocación a la función y pueden ser expresiones del mismo tipo del parámetro formal. Los parámetros deben colocarse en la misma posición del parámetro formal correspondiente.

2.4 PRE y POST condición:

Si asumimos que un programa necesita que la entrada tenga un valor especifico, se dice que esa es la **precondición** del programa la **poscondición** se refiere al estado después de la ejecución del programa.

Por ejemplo:

```
f = raíz(X);      /* PRE: X>=0 */
                  /* POST: Y = √x */
```

La pre y post condición se colocan junto con el prototipo de la función en el archivo header. Pues estas sirven de información al programador que usa la función.