Universidad Tecnológica Nacional
Facultad Regional Córdoba
Dpto de Ingeniería en Sistemas de Información
Cátedra de Paradigmas de Programación

UNIDAD Nº 2

PARADIGMA IMPERATIVO

EL PRESENTE CAPÍTULO HA SIDO ELABORADO, CORREGIDO Y COMPILADO POR:

CORSO, CYNTHIA
GUZMAN, ANALIA
LIGORRÍA, LAURA
MARCISZACK, MARCELO
TYMOSCHUCK, JORGE

ÍNDICE

OBJETIVOS DE LA UNIDAD

Que el alumno, comprenda las ventajas de la separación de código mediante el uso de funciones, para facilitar su construcción y verificación de un programa.
Que el alumno, aprecie las ventajas de la declaración de variables locales y globales, para entender la necesidad de contar con mayor control sobre la accesibilidad de los datos en un programa.

CONTENIDOS ABORDADOS

Introducción.
Programación estructurada.
Características generales: Variables locales y globales. Estructuras de control. Modularización (funciones y procedimientos).
Introducción al lenguaje C: Tipos de datos. Declaración de variables. Operadores. Procedimientos. Funciones. Estructuras de control. Funciones de entrada y salida.

1. INTRODUCCION

El paradigma de la programación imperativa se basa en resolver los problemas a partir de la ejecución de programas que indican a la computadora un conjunto de acciones precisas a realizar. En este paradigma, la ejecución de los programas se limita a procesar cada instrucción en el orden en que se han escrito. En otras palabras, al construir un programa se indica a la computadora “como” realizar el proceso.

Inicialmente los programas se apoyaban en una sola instrucción para modificar la secuencia de ejecución de las instrucciones mediante una transferencia incondicional de su control (con la instrucción goto, del inglés "go to", que significa "ir a"). A este tipo de programación también se lo conoce como programación lineal o no estructurada.

Una instrucción "goto" permite pasar el control a cualquier otra parte del programa. Cuando se ejecuta una instrucción "goto" la secuencia de ejecución del programa continúa a partir de la instrucción indicada por "goto". De esta forma, para comprender como funciona un programa es necesario simular su ejecución. Esto quiere decir que en la mayoría de los casos es muy difícil comprender la lógica de un programa de este tipo. Lenguajes, como Fortran, están basados en el uso de la instrucción goto.

Pero estas transferencias arbitrarias del control de ejecución hacían los programas muy poco legibles y difíciles de comprender. A finales de los años sesenta, surgió una nueva forma de programar que reducía a la mínima expresión el uso de la instrucción goto y la sustituyó por otras más comprensibles.

Así surgió la programación estructurada, que nació como solución a los problemas que presentaba la programación no estructurada, esta forma de programar se basa en un famoso teorema llamado “El **teorema del programa estructurado” que** es un resultado en la teoría de lenguajes de programación y establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) específicamente son:

- **Secuencia**: ejecución de una instrucción tras otra.
- **Selección**: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- **Iteración**: ejecución de una instrucción (o conjunto) mientras una variable booleana sea 'verdadera'. Esta estructura lógica también se conoce como ciclo o bucle.

Este teorema demuestra que la instrucción GOTO no es estrictamente necesaria y que para todo programa que la utilice existe otro equivalente que no hace uso de dicha instrucción.

Los científicos de la computación usualmente acreditan el teorema a un artículo de 1966 escrito por Corrado Böhm y Giuseppe Jacopini. Sin embargo, David Harel rastreó sus orígenes hasta la descripción de 1946 de la arquitectura de von Neumann y el teorema de la forma normal de Kleene. La demostración de Böhm-Jacopini describe cómo construir diagramas de flujo estructurados a partir de cualquier digrama de flujo, usando los bits de una variable entera extra para dar seguimiento a la información que el programa original representa mediante puntos de entrada en el código. Esta construcción estuvo basada en el lenguaje de programación P″ de Böhm. La demostración de Böhm-Jacopini no esclareció la cuestión sobre cuándo convendría usar programación estructurada para el desarrollo de software, en parte porque la construcción ofuscaba el código del programa en lugar de mejorarlo. Por otro lado, fue el punto de partida para iniciar el debate. Edsger Dijkstra escribió una importante carta titulada "La sentencia Go To considerada dañina" en el año 1968. Posteriores estudios agregaron aproximaciones más prácticas a la demostración de Böhm-Jacopini, que mantenían o mejoraban la claridad del programa original.

1.1 PROGRAMACIÓN ESTRUCTURADA

La programación estructurada tiende a ser ***orientada a* la *acción.*** Los programadores se **concentran en escribir procesos (procedimientos o funciones)**, que son grupos de acciones que ejecutan alguna tarea común y que se agrupan para formar programas. La unidad fundamental de la programación es la acción, no las entidades como en la Programación orientada a objetos. Es cierto que los datos son importantes, pero la óptica es que los datos son materia prima para las acciones que los procesos ejecutan.

Se basa en una metodología de desarrollo de programas llamada refinamientos sucesivos: Se plantea una operación como un todo y se divide en segmentos más sencillos o de menor complejidad. Una vez terminado todos los segmentos del programa, se procede a unificar las aplicaciones realizadas por el pool de programadores. Si se ha utilizado adecuadamente la programación estructurada, esta integración debe ser sencilla y no presentar problemas al integrar la misma, y de presentar algun problema, será rápidamente detectable para su corrección.

La representación gráfica se realiza a través de diagramas de flujo o flow chart, el cual representa el programa con sus entradas, procesos y salidas.

Propone segregar los procesos en estructuras lo más simple posibles, las cuales se conocen como secuencia, selección e interacción. Ellas están disponibles en todos los lenguajes modernos de programación imperativa en forma de sentencias. Combinando esquemas sencillos se pueden llegar a construir sistemas amplios y complejos pero de fácil entendimiento.

Es una técnica en la cual la estructura de un programa, esto es, la interpelación de sus partes realiza tan claramente como es posible mediante el uso de tres estructuras lógicas de control:

Secuencia: Sucesión simple de dos o mas operaciones.

Selección: bifurcación condicional de una o mas operaciones.

Iteración: Repetición de una operación mientras se cumple una condición.

Estos tres tipos de estructuras lógicas de control pueden ser combinados para producir programas que manejen cualquier tarea de procesamiento de información.

2. CARACTERÍSTICAS GENERALES

2.1 VARIABLES LOCALES Y GLOBALES

Las variables son el espacio de *almacenamiento* en memoria donde los programas imperativos guardan valores que utilizan en los procesos que realizan. Se dividen en dos grandes grupos según su ámbito.

2.1.1 VARIABLES LOCALES

Son aquellas que se declaran y son accesibles en un contexto determinado. Al finalizar dicho contexto (llamado ámbito) la variable se destruye perdiendo su valor y toda posibilidad de ser utilizada.

2.1.2 VARIABLES GLOBALES

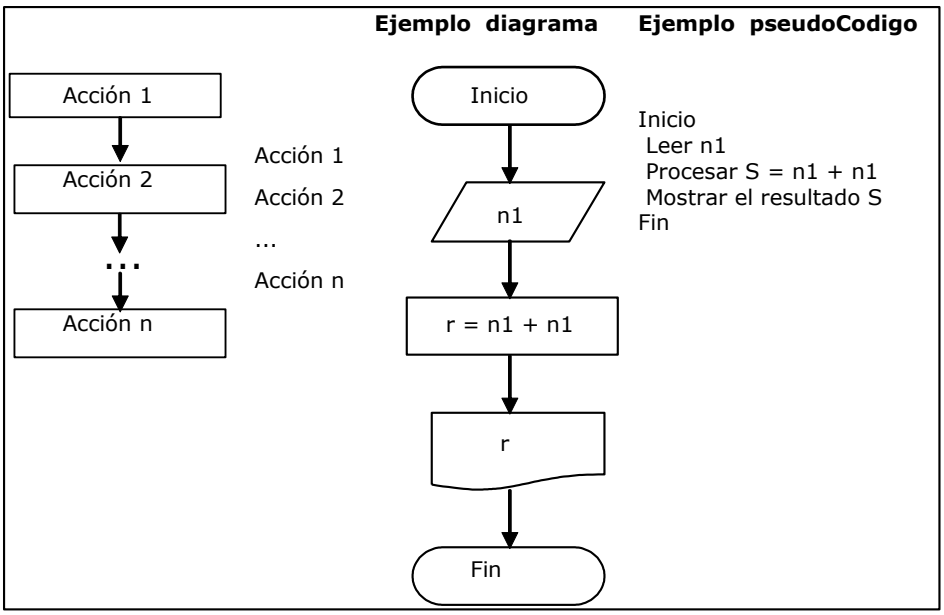
Son aquellas que perduran en un ámbito mayor, por ejemplo, todo el programa. Son accesibles desde cualquier parte del programa y almacenan un valor perdurable que trasciende cualquier ámbito.

2.2 ESTRUCTURAS DE CONTROL

Es la manera como se van encadenando, uniendo entre sí las acciones, dando origen a los distintos tipos de estructuras.

2.2.1 ESTRUCTURAS SECUENCIALES

La estructura secuencial es aquélla en la que una acción sigue a otra en secuencia.



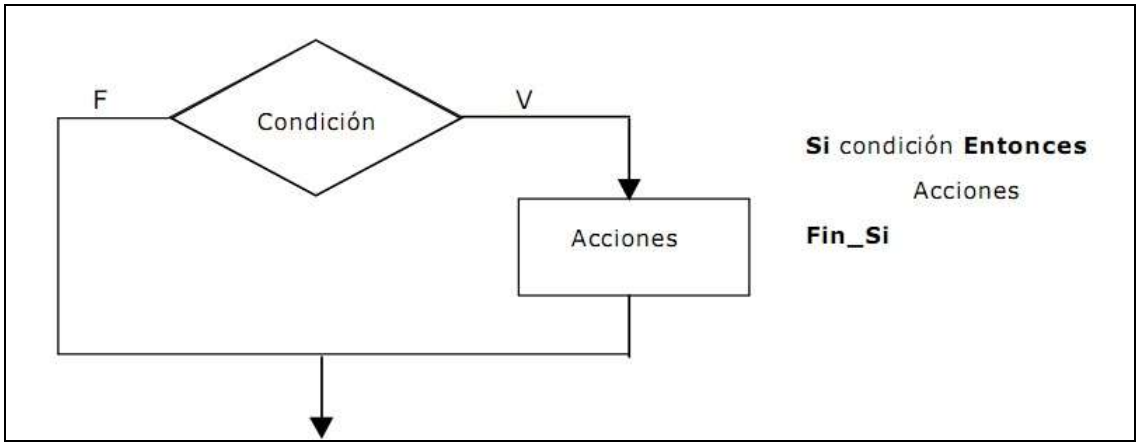
2.2.2 ESTRUCTURAS CONDICIONALES, SELECTIVAS O ALTERNATIVAS

Son estructuras de control que permiten evaluar condiciones y ejecutar acciones en función del resultado que arrojen estas condiciones.

2.2.1 .1 CONDICIONAL SIMPLE

Permite evaluar una condición y ejecuta un bloque de acciones solo en el caso en que dicha condición arroje un resultado verdadero. En caso contrario, no se ejecuta ninguna operación.

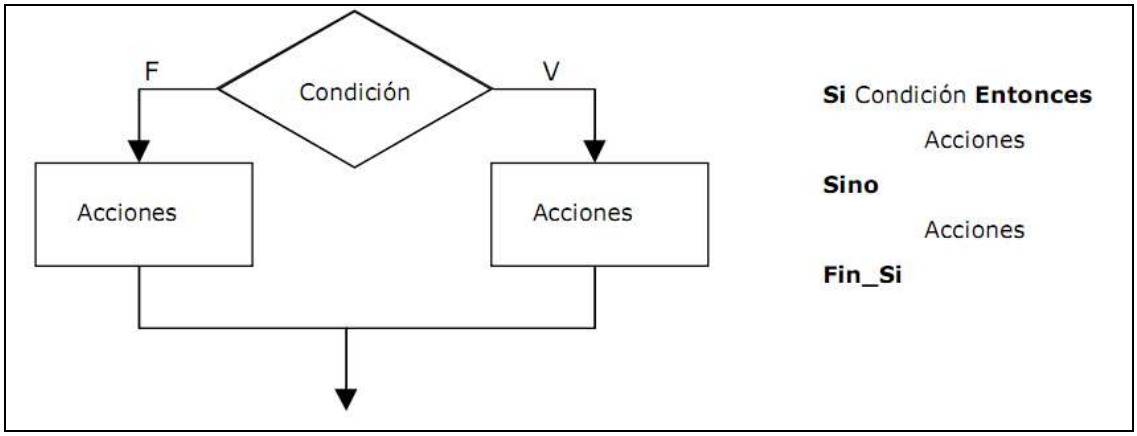
Gráficamente, mediante diagrama de flujo:



2.2.1.2 CONDICIONAL DOBLE

Permite evaluar una condición y tanto para el caso de arrojar un valor verdadero como falso ejecuta bloques de acciones.

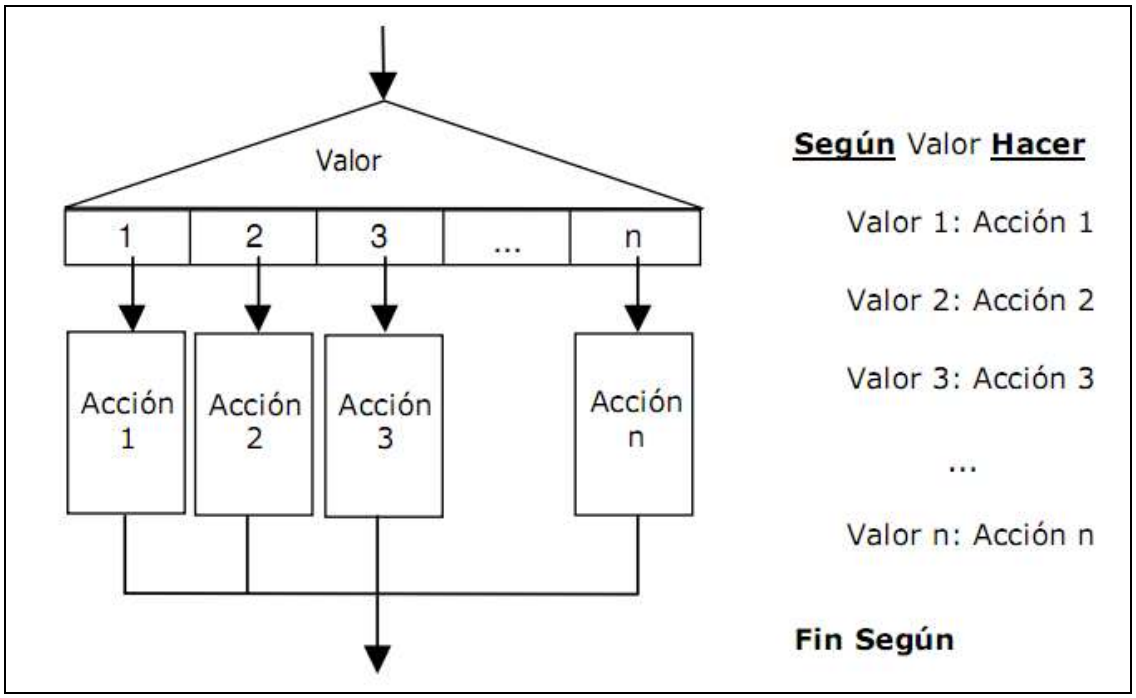
Gráficamente:



2.2.1.3 CONDICIONAL MÚLTIPLE

Permite comparar una variable contra varios valores diferentes permitiendo ejecutar un conjunto de operaciones para cada valor posible.

Gráficamente, mediante diagrama de flujo:



2.2.3 ESTRUCTURAS REPETITIVAS O ITERATIVAS

Estas estructuras de control permiten repetir un conjunto de instrucciones y suelen llamarse ciclos, bucles, lazos, etc.

Un ciclo es un segmento de un algoritmo o programa, cuyas instrucciones se repiten un número determinado de veces mientras se cumple una determinada *condición.*

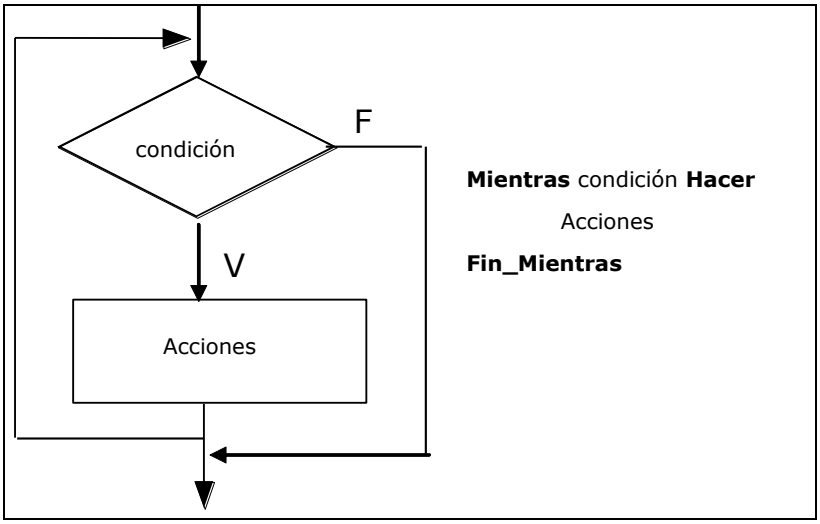
Un ciclo consta de tres partes:

- Decisión o condición.
- Cuerpo del ciclo.
- Salida del ciclo.

2.2.2.1 CICLO WHILE

Permite evaluar una condición y repetir un conjunto de operaciones mientras la condición siga arrojando resultado verdadero cada vez que es evaluada. Ciclo 0 a N

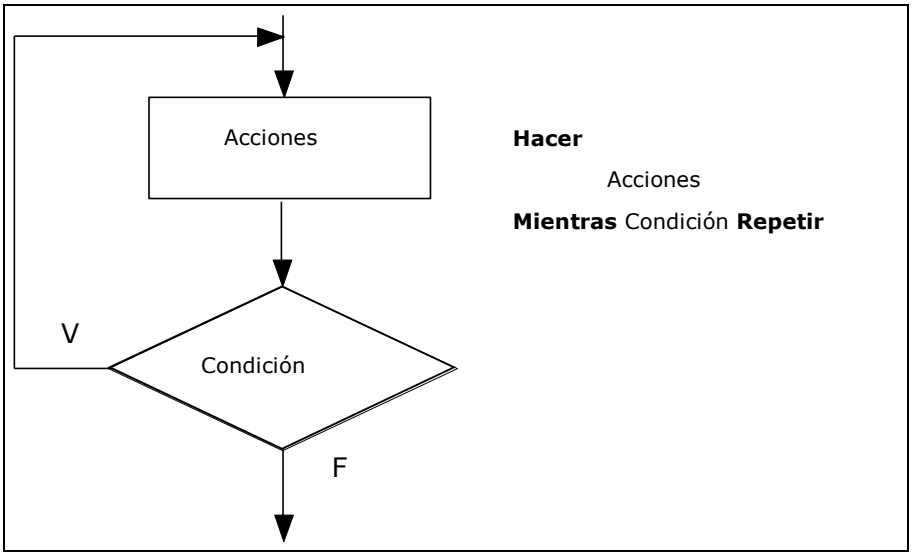
Gráficamente, mediante diagrama de flujo:



2.2.2.2 CICLO DOWHILE

Funciona de una manera muy similar al ciclo while pero el bloque de acciones se ejecuta antes de avaluar la condición. Dicho de otra forma, primero se ejecutan las acciones y luego se evalúa la condición. El ciclo deja de repetir cuando el resultado de la condición es falso.

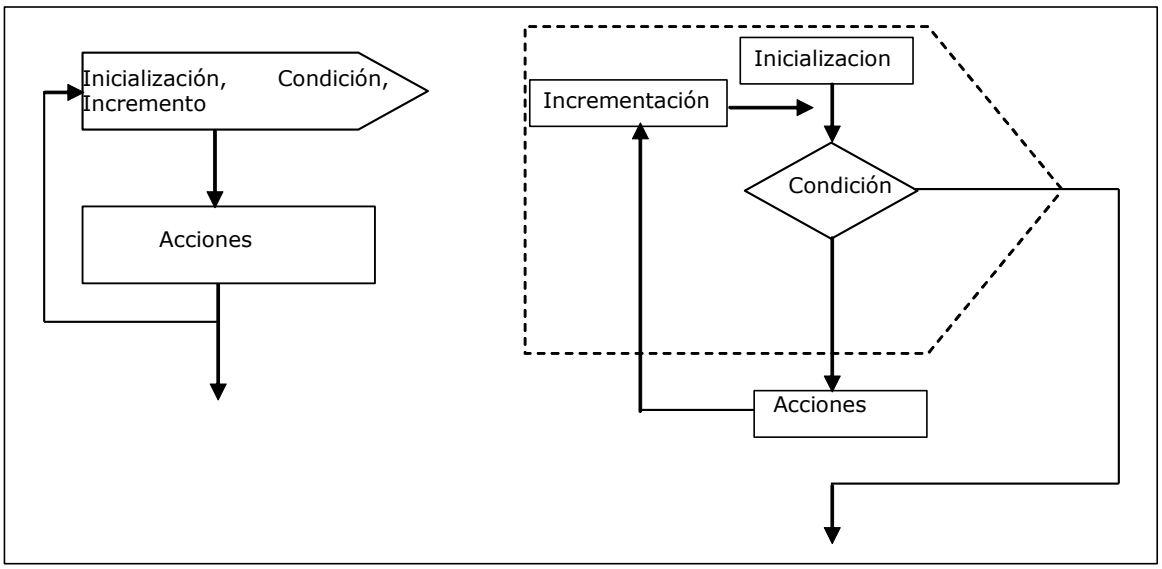
Gráficamente, mediante diagrama de flujo:



2.2.2.3 CICLO FOR

Permite repetir un grupo de acciones un número determinado de veces. Su utilización se ajusta especialmente a situaciones en las que se conoce la cantidad de iteraciones que se desea realizar.

Gráficamente, mediante diagrama de flujo:



2.3 MODULARIZACIÓN

Un módulo es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema complejo original. Cada uno de estos módulos tiene una tarea bien definida y algunos necesitan de otros para poder operar. En caso de que un módulo necesite de otro, puede comunicarse con éste mediante una interfaz de comunicación que también debe estar bien definida.

Si bien un módulo puede entenderse como una parte de un programa en cualquiera de sus formas y variados contextos, en la práctica se los suele tomar como sinónimos de procedimientos y funciones. Pero no necesaria ni estrictamente un módulo es una función o un procedimiento, ya que el mismo puede contener muchos de ellos, no debe confundirse el término "modulo" (en el sentido de programación modular) con términos como "función" o "procedimiento", propios del lenguaje que lo soporte.

Particularmente, en el caso de la programación, los módulos suelen estar (no necesariamente) organizados jerárquicamente en niveles, de forma que hay un módulo principal que realiza las llamadas oportunas a los módulos de nivel inferior.

Cuando un módulo es convocado, recibe como entrada los datos proporcionados por otro del mismo o superior nivel, el que ha hecho la llamada; luego realiza su tarea. A su vez este módulo convocado puede llamar a otro u otros módulos de nivel inferior si fuera necesario; cuando ellos finalizan su tarea, devuelven la salida pertinente al módulo inmediato llamador, en secuencia reversa, finalmente se continúa con la ejecución del módulo principal.

Características de un módulo

Cada uno de los módulos de un programa idealmente debería cumplir las siguientes características:

- Tamaño relativamente pequeño: Esto facilita aislar el impacto que pueda tener la realización de un cambio en el programa, bien para corregir un error, o bien por rediseño del algoritmo correspondiente.
- Independencia modular: Cuanto más independientes son los módulos entre sí más fácil y flexiblemente se trabajará con ellos, esto implica que para desarrollar un módulo no es necesario conocer detalles internos de otros módulos. Como consecuencia de la independencia modular un módulo cumplirá:
    - Características de caja negra, es decir abstracción Aislamiento de los detalles mediante encapsulamiento
    - La independencia modular mejora el rendimiento humano, pudiendo realizarse programación en equipo y desarrollar módulos paralelamente. También contribuye a la reutilización de software.

2.3.1 PROCEDIMIENTOS Y FUNCIONES

En computación, una **subrutina** o **subprograma** (también llamada **procedimiento**, **función** o **rutina**), es una porción de código que realiza una tarea específica y forma parte de un programa más grande. Se presenta como un subalgoritmo que forma parte del algoritmo principal.

Una subrutina al ser llamada dentro de un programa hace que el código principal se detenga y se dirija a ejecutar el código de la subrutina, cuando esta termina el código principal continua con su ejecución.

La mayoría de los lenguajes de programación soportan la creación de subrutinas y métodos para llamarlas (invocarlas) y retornarlas.

Existen varias ventajas de "romper" un programa en subrutinas:

- Reducción de código duplicado.

- Descomposición de problemas complejos en simples piezas (lo que aumenta la mantenibilidad y extensibilidad).
- Aumenta la legibilidad del código de un programa.
- Permite la reutilización de código en múltiples programas.

Diagrama del funcionamiento de una subrutina.

Las declaraciones de subrutinas generalmente son especificadas por:

- Un nombre único en el ámbito: nombre con el que se identifica y se distingue de otras. No podrá haber otra con ese nombre (salvo sobrecarga o polimorfismo en programación orientada a objetos).
- Un tipo de dato de retorno: tipo de dato del valor que la subrutina devolverá al terminar su ejecución.
- Una lista de parámetros: especificación del conjunto de argumentos (pueden ser cero, uno o más) que la función debe recibir para realizar su tarea.
- El código u órdenes de procesamiento: conjunto de órdenes y sentencias que debe ejecutar la subrutina.

Las subrutinas se pueden clasificar de dos formas:

- **Procedimiento**: es un subproceso que puede recibir o no parámetros de entrada o de entrada / salida y realiza un conjunto de acciones sin retornar valores.
- **Función**: es un subproceso que puede recibir o no parámetros de entrada o de entrada / salida y realiza un conjunto de acciones retornando un único resultado.

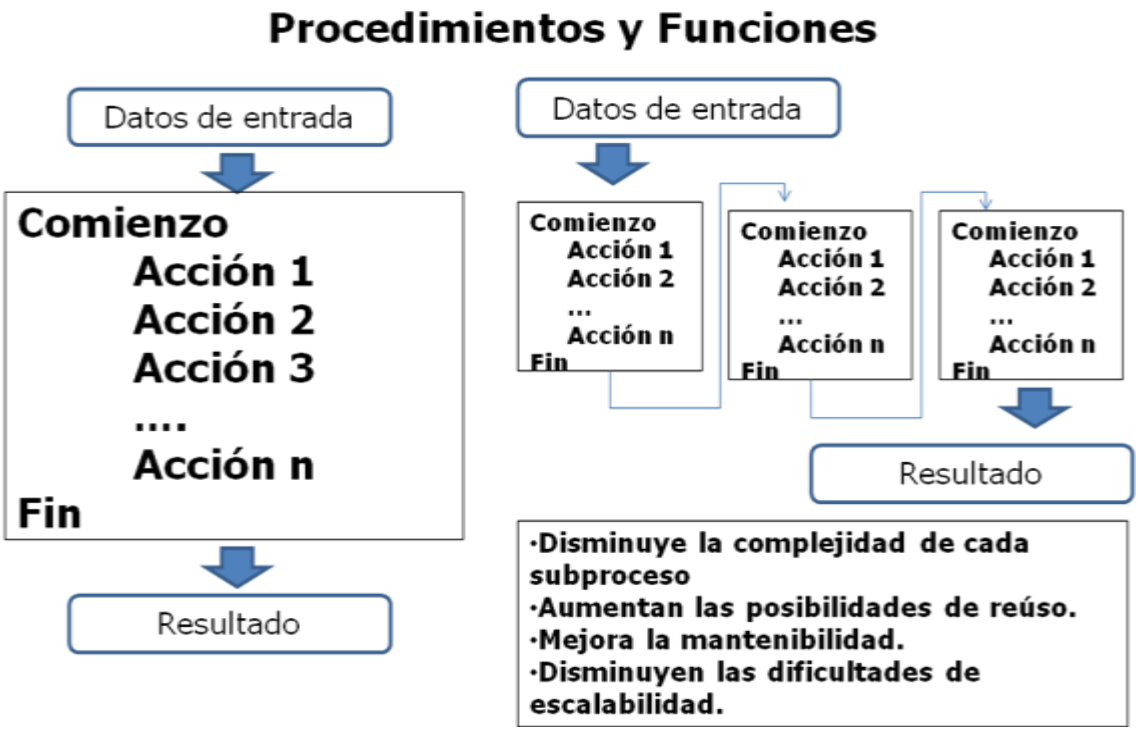
En varios lenguajes de programación imperativa, el concepto de procedimiento y función ha sido unificado en un único recurso técnico. Por ejemplo en lenguaje C solo existen las funciones y los procedimientos se programan como funciones que no retornan ningún valor. En el lenguaje Pascal distingue entre la subrutina Procedimiento y la subrutina Función. En cambio existen otros lenguajes que los tratan como sinónimos.

La imagen siguiente nos muestra a la derecha un procedimiento y a la izquierda una función.

L

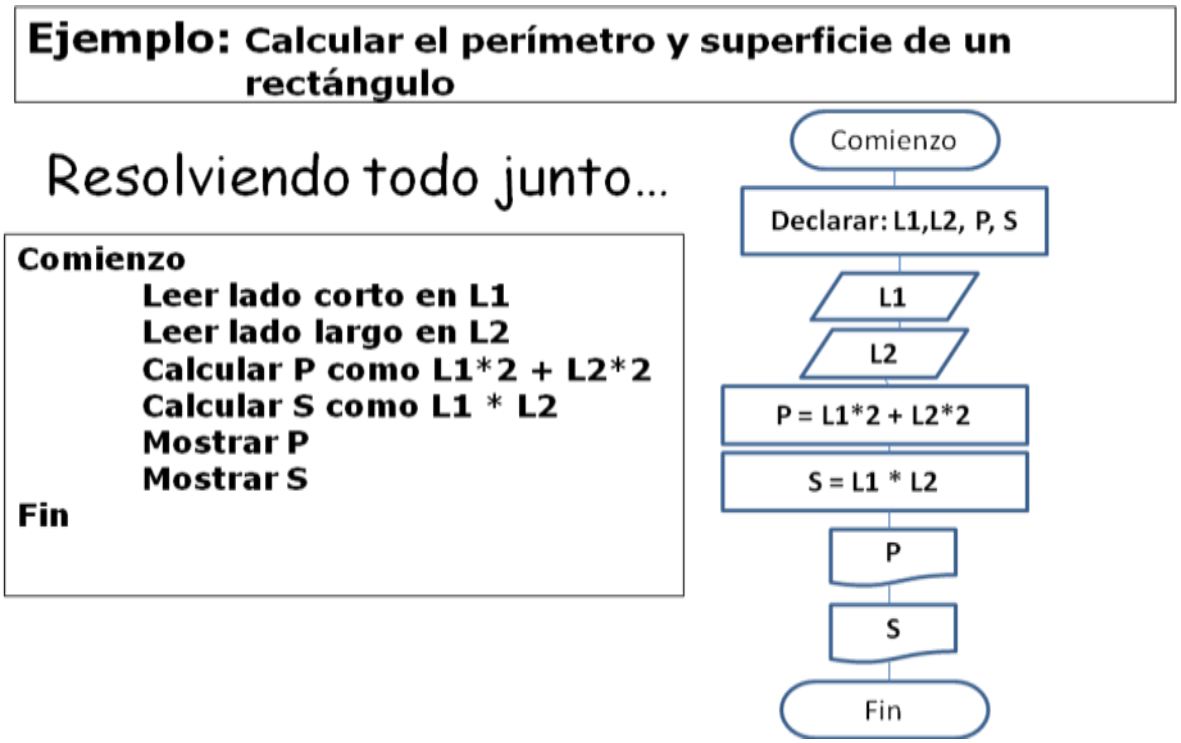
La función **también puede retornar valores sobre los parámetros de entrada**, si son de tipo referencia.

La imagen siguiente nos muestra a la izquierda la estructura de un programa imperativo resuelto de manera lineal, o sea, sin divisiones en subprocesos (procedimientos o funciones). A la derecha se encuentra el mimo programa pero desarrollado con división en subprocesos.

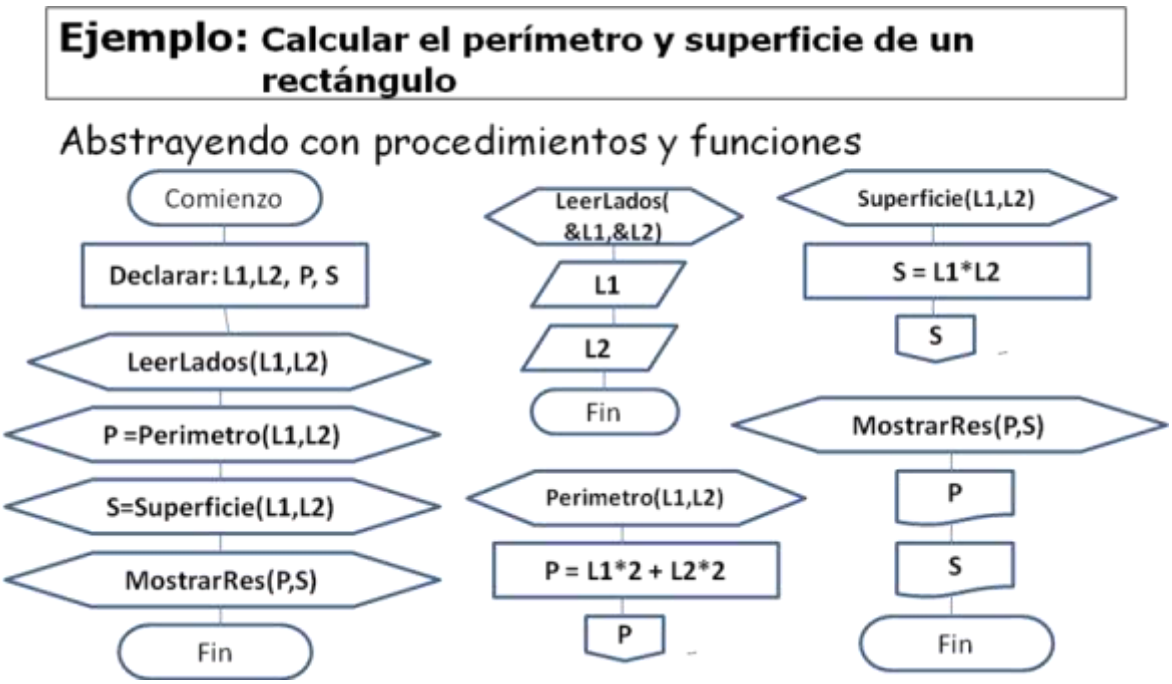


3. EJEMPLO DE UN PROGRAMA IMPERATIVO

Veamos un primer ejemplo de un programa imperativo, primero sin dividir en subprocesos y luego identificando procedimientos y funciones.



Y ahora subdividido…



4. ANEXO I: INTRODUCCIÓN A LENGUAJE C

Existen varios lenguajes de programación imperativa que han tenido mayor o menor evolución en el tiempo. Ejemplos de estos pueden ser: Pascal, Cobol, Fortram, RPG, C, etc.

Durante esta unidad utilizaremos lenguaje C, por lo que presentamos a continuación su estructura y sintaxis básica. Seguramente el lector notará una amplia similitud con lenguaje Java, C++, C#, PHP y otros. Esto se debe a que todos estos tomaron de C la sintaxis general con el objeto de suavizar la curva de aprendizaje de los miles de programadores que utilizaban C y de esa forma lograr una comunidad importante en menor tiempo.

4.1 TIPOS DE DATOS EN C

La siguiente es una planilla que muestra los tipos de datos existentes en lenguaje C

TIPO	ANCHO EN BIT	RANGO EN PC
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long double	64	1.7E-308 a 1.7E+308

4.2 DECLARACIÓN DE VARIABLES EN C

En lenguaje C, la forma general de declaración de una variable es “tipo nombre;”. O sea, tipo de datos, espacio, nombre de variable y punto y coma. Si se van a declarar muchas variables del mismo tipo se pueden separar por comas si volver a escribir el tipo de dato.

Por ejemplo, para declarar una variable entera llamada x sería:

```
int x;
```

y si es char sería:

```
char x;
```

4.3 OPERADORES EN C

La figura siguiente muestra una lista de los operadores más importantes existentes en lenguaje C.

SIMBOLO	DESCRIPCION	EJEMPLO
=	**asignación**	**a = b**
+	suma	a + b
-	resta	a - b
*	multiplicación	a * b
/	división	a / b
%	modulo	a % b
-	negación	-a

4.4 DECLARACIÓN DE PROCEDIMIENTOS EN C

En lenguaje C no existen los procedimientos, todo se implementa mediante funciones. Existe un tipo de datos llamada void (vacío). Cuando se declara que una función tiene un tipo de retorno void, se comporta como un procedimiento ya que no retorna resultado.

Forma general:

```
void nombre(lista parámetros)
```

Ejemplo:

```
void miProcedimiento(int p1, char p2, float p3)
```

4.5 DECLARACIÓN DE FUNCIONES EN C

Forma general:

```
tipoRetorno nombre(lista parámetros)
```

Ejemplo:

```
int miFuncion(int p1, char p2, float p3)
```

4.6 CONDICIONALES EN C

Forma genera condicional simple

```
If(condición)
{
        Accion1;
        Accion2;
        …
}
```

Forma general condicional doble

```
If(condición)
{
        Accion1;
        Accion2;
        …
}
else
{
        Accion1;
        Accion2;
        …
}
```

4.7 ITERACIONES EN C

Forma general del ciclo while

```
While(condición)
{
        Accion1;
        Accion2;
        …
}
```

Forma general del ciclo do while

```
do
{
        Accion1;
        Accion2;
        …
}while(condición);
```

Forma general del ciclo for

```
for(operacion1; condición; operacion2)
{
        Accion1;
        Accion2;
        …
}
```

Aclaración: El ciclo for de lenguaje C es igual al de Java. Lo que en la forma general se ha escrito como operacion1, puede ser cualquier expresión, incluso una llamada a una función para que realice un bloque de acciones. Solo hay que tener en cuenta que esta operación se realiza primero y por única vez, por eso se la utiliza muy frecuentemente para inicializar una variable que sirva para contar la cantidad de vueltas del ciclo.

Luego de realizar la operación1, se ejecuta la condición y si da verdadera el ciclo realiza su primera iteración.

Al finalizar la primera iteración se realiza la operacion2 y luego se evalúa nuevamente la condición. Si la misma da resultado verdadero el ciclo vuelve a repetir hasta que entregue un resultado falso. Como la operación2 se realiza una vez al finalizar cada iteración, muy frecuentemente se la utiliza para incrementar el valor de una variable que cuenta la cantidad de vueltas del ciclo, pero en realidad (al igual que operacion1) puede contener casi cualquier instrucción, incluso llamado a funciones.

4.8 FUNCIONES DE ENTRADA / SALIDA EN C

Para mostrar resultados y literales en pantalla se utiliza la función printf que recibe una lista de parámetros separados que normalmente son literales o varialbes.

Forma general printf

```
printf(parametro1, parametro2, …);
```

Ejemplo printf

```
printf(“El valor de x es:  “, x);
```

Para leer datos de teclado se utiliza la función scanf que recibe dos parámetros. El primero indica el tipo de datos de lo que se pretende leer y el segundo es la variable sobre la que se desea hacer la lectura.

Forma general scanf

```
scanf(tipo, &variable);
```

Ejemplo scanf:

```
// Este ejemplo guarda un número en n.

int n;
printf("Introduce un numero: ");
scanf("%d",&n);

// Este ejemplo guarda un caracter en m.

char m;
printf("Introduce un caracter: ");
scanf("%c",&m);

// Este ejemplo guarda una cadena de caracteres (solamente una
palabra) en cad.
// Notese la ausencia de &

char cad[20];
printf("Introduce una palabra: ");
scanf("%s",cad);

printf("Introduce una palabra: ");
scanf("%10s",cad);  // lee máximo 10 caracteres y le concatena el
caracter cero.
```

5. EJERCICIOS PRÁCTICOS

Ej1: Se desea desarrollar un sencillo programa que teclee un número entero n, después visualizar la lista de enteros comprendidos entre 1 y n, y calcula el total de números visualizados.

Resolución:

```
#include<stdio.h>
int leer_numero(void){
     int n;
     /* lectura del número*/
     printf(" Escribir un numero: ");
     scanf("%d", &n);
     return n;
}
void main(void){
     int i, fin;
     long total = 0;
     fin = leer_numero();
     for(i=1;i<=fin;i++){
        total+=i;
        printf("i= %d", i);
        printf("***total= %ld\n", total);
     }
     printf("***total= %ld\n", total);
```

```
     getch();
}/*fin del main*/
```

Ej2: Se desea saber cual es el sueldo de los empleados que trabajan una cierta cantidad de horas, sabiendo que se paga $30 la hora de trabajo. Cargando el legajo y las horas trabajadas de cada empleado, se pide:

- ✓ El importe del mayor sueldo y el legajo de quien lo obtuvo
- ✓ Calcular la diferencia entre el promedio y el menor sueldo
- ✓ Si la diferencia del punto anterior es menor a $250, calcular un aumento del 15% sobre el menor sueldo. Mostrar el monto de ese sueldo y el legajo del empleado correspondiente

Fin de la carga: legajo = 0

Resolución:

```
#include<stdio.h>

void main(void)
{
    int leg, horas, contador_s, leg_may, leg_men, bPrimero;
    float sueldo,may, men, dif, sum_s,promedio;
    men = may = sum_s = contador_s = 0;
    bPrimero = 0;

    /* Primera lectura de legajos*/
    printf("\n Ingrese legajo: ");
    scanf("%d", &leg);

    while(leg != 0)
    {
      printf("\n Ingrese cantidad de horas trabajadas: ");
      scanf("%d", &horas);
      sueldo = (horas * 30);
      system("cls");

      if(bPrimero == 0){
      /* Para la primer lectura el sueldo es mayor y menor
      simultáneamente*/
         may = men= sueldo;
         leg_may = leg_men = leg;
         bPrimero=1;
      }
      else{
           if(sueldo > may)
          {
            may = sueldo;
            leg_may = leg;
          }
          else
```

```
            {
               if(sueldo < men)
                {
                  men = sueldo;
                  leg_men = leg;
                }
            }
       }
       sum_s = sum_s + sueldo;
       contador_s = contador_s  + 1;
       printf("\n Ingrese legajo: ");
       scanf("%d", &leg);
    }
    if(contador_s>0){
        promedio = (sum_s / contador_s);

        dif = promedio - men;
        if(dif > 250)
        {
          men= ((15 * men)/100);
          printf("\n El nuevo sueldo es: %f", men);
          printf(" correspondiente al legajo: %d", leg_men);
        }
      printf("\n El mayor sueldo es: %f correspondiente al trabajador
      cuyo legajo es: %d",may, leg_may);

      printf("\n La diferencia entre el menor sueldo y el sueldo
      promedio es de: %f",dif);
    }
    else{
         printf("\n Sin datos de legajos procesados");
    }
    getch();
}/*fin del main*/
```

Ej3: Una empresa se dedica a la fabricación de tres tipos diferentes de artículos para oficina. Sabiendo el código de artículo, la cantidad vendida (de ese artículo), el tipo y el precio, la empresa necesita un programa que le permita conocer:

- El tipo de producto que mas se vendió
- El importe total vendido por cada tipo de articulo (tres totales)
- El importe promedio de venta por cada artículo
- Si el importe del artículo supera al promedio en $400 calcular un aumento del 7% en el precio unitario del artículo y mostrar el nuevo precio.

Fin de la carga: código de artículo = -1

Resolución:

```
#include<stdio.h>
void main(void)
```

```
{
     /*Declaración de variables*/
     int cod_art, canti_ven, tipo_art, c1,c2,c3, tip_may;
      float precio, importe, sum1, sum2, sum3, promedio1, promedio2,
      promedio3;
     /*Inicialización de contadores y sumadores a cero*/
     c1=c2=c3=sum1=sum2=sum3=0; //

    printf("\n Ingrese codigo de articulo: ");
    scanf("%d", &cod_art);
    while(cod_art != -1)
    {
       printf("\n ingrese cantidad vendida: ");
       scanf("%d", &canti_ven);
       printf("\n Ingrese tipo de producto: ");
       scanf("%d", &tipo_art);
       printf("\n Ingrese precio: ");
       scanf("%f", &precio);
       /*Limpiamos la pantalla*/
       system("cls");

       importe= canti_ven*precio;

       switch(tipo_art)
       {
             case 1: {
                   c1++;
                   sum1+=importe;
                   break;
                 }
             case 2: {
                   c2++;
                   sum2+=importe;
                   break;
                 }
             case 3: {
                   c3++;
                   sum3+=importe;
                   break;
                }
           default:
                   printf("\n\a Error. Tipo de articulo ingresado no
           existente");
       }/*Fin switch*/

       printf("\n Ingrese codigo de articulo: ");
       scanf("%d", &cod_art);
   }/*fin del while*/

   if(c1>0)
```

```
        promedio1=sum1/c1;
    else
        promedio1=0;
    if(c2>0)
        promedio2=sum2/c2;
    else
        promedio2=0;

    if(c3>0)
        promedio3=sum3/c3;
    else
        promedio3=0;

  if(c1>c2 && c1>c3)
      tip_may = c1;
  else
        if(c2 > c3)
              tip_may = c2;
        else
              tip_may = c3;

    /*****RESULTADOS*/
    printf("Los importes totales vendidos por cada tipo de articulo
    son: %f %f %f respectivamente\n",sum1,sum2,sum3);
    printf("Los importes promedio de venta son: %f %f %f
    respectivamente\n",promedio1,promedio2,promedio3);
    printf("El tipo de articulo mas vendido fue: %d",tip_may);
    getch();
}/*fin del main*/
```

6. BIBLIOGRAFÍA

- Joyanes Aguilar, Luis – 2006 – Programación en C++ Algoritmos, Estructuras de Datos y Objetos. Ed. McGraw Hill.

- Brian W. kernighan/ Dennis M. Ritchie - 1978 - El lenguaje de programación C . Ed. Pearson Educación