Violeta Ocegueda
Profesor-Investigador
Facultad de Ciencias Químicas e Ingeniería
Universidad Autónoma de Baja California
Campus Tijuana

Programación

Tronco Común de Ingeniería

FCQI

Contenido

Metodología para la resolución de problemas

Metodología para la resolución de problemas

Problema

Un problema se entiende como una proposición que, a partir de ciertas condiciones conocidas, induce a buscar algo desconocido.

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es -esencialmente- un **proceso creativo**, se pueden considerar una serie de fases o pasos comunes a seguir.

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Metodología para la resolución de problemas

Las fases de resolución de un problema con computadora son:

- Análisis del problema
- Diseño del algoritmo
- Codificación
- Compilación y ejecución
- Verificación
- Depuración
- Mantenimiento
- Documentación

Análisis del problema

- Es la primera fase de la resolución de un problema con computadora.
- Requiere una clara definición de las **entradas** y **salidas**.

Ejercicio: Describa el proceso de retirar dinero del cajero.

Diseño del algoritmo

Algoritmo

Un algoritmo es un conjunto de pasos, procedimientos o acciones que nos permiten alcanzar un resultado o resolver un problema.

Un algoritmo se puede concebir como un diálogo entre una computadora y una persona, en el que se especifica bajo qué condiciones la computadora debe generar la salida específica.

Características de los algoritmos

Las características que los algoritmos deben reunir son las siguientes:

- Precisión: los pasos a seguir en el algoritmo deben ser precisados claramente.
- Determinismo: dado un conjunto de datos idénticos de entrada, siempre debe arrojar los mismos resultados.
- Finitud: independientemente de su complejidad, siempre debe ser de longitud finita.

Características de los algoritmos

Las características que los algoritmos deben reunir son las siguientes:

- Precisión: los pasos a seguir en el algoritmo deben ser precisados claramente.
- Determinismo: dado un conjunto de datos idénticos de entrada, siempre debe arrojar los mismos resultados.
- Finitud: independientemente de su complejidad, siempre debe ser de longitud finita.

Características de los algoritmos

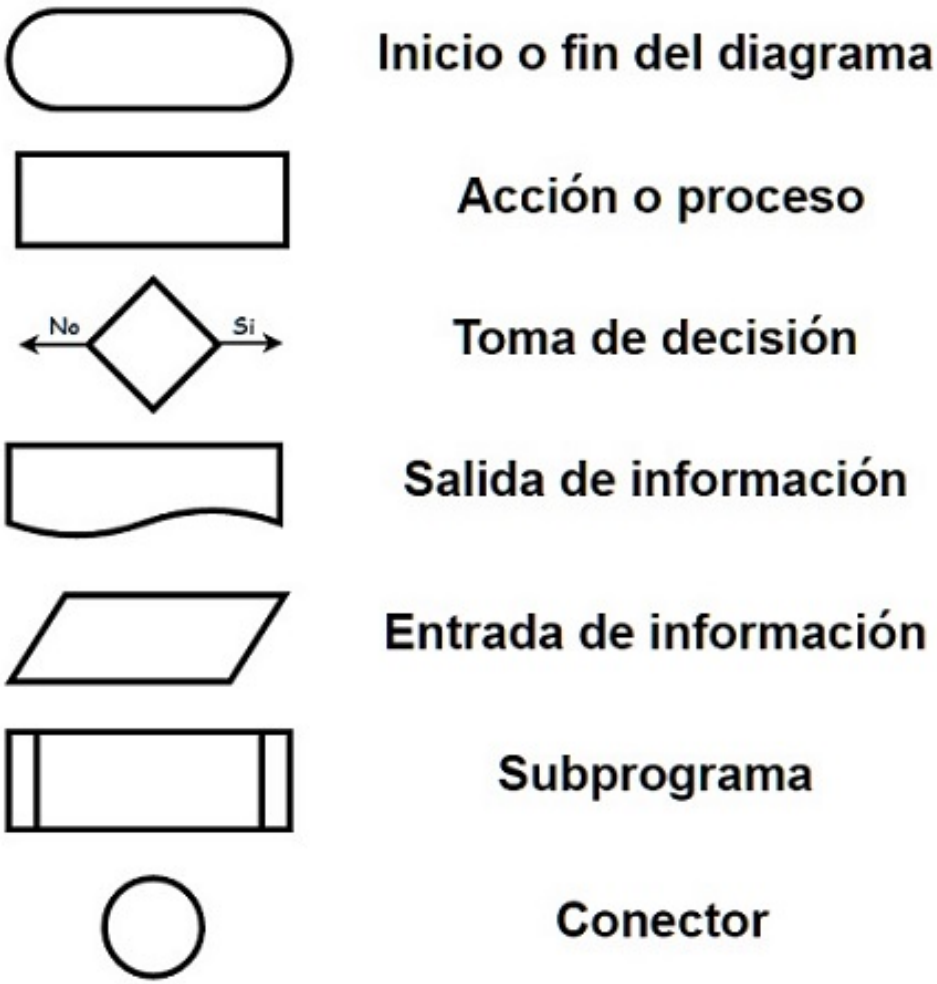
Las características que los algoritmos deben reunir son las siguientes:

- Precisión: los pasos a seguir en el algoritmo deben ser precisados claramente.
- Determinismo: dado un conjunto de datos idénticos de entrada, siempre debe arrojar los mismos resultados.
- Finitud: independientemente de su complejidad, siempre debe ser de longitud finita.

Diagrama de flujo

Un diagrama de flujo representa la esquematización gráfica de un algoritmo. Es decir, muestra gráficamente los pasos o procesos a seguir para alcanzar la solución de un problema.

Símbolos utilizados en los diagramas de flujo



Codificación

Codificación es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en etapas anteriores.

Depuración

La depuración es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.
Cuando se ejecuta un programa se pueden producir tres tipos de errores:

- Errores de compilación: o *errores de sintaxis*, se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación.
- Errores de ejecución: se producen por instrucciones que la computadora puede comprender pero no ejecutar (división por cero, raíces cuadradas de números negativos).
- Errores lógicos: se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo.

Depuración

La depuración es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.
Cuando se ejecuta un programa se pueden producir tres tipos de errores:

- Errores de compilación: o *errores de sintaxis*, se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación.
- Errores de ejecución: se producen por instrucciones que la computadora puede comprender pero no ejecutar (división por cero, raíces cuadradas de números negativos).
- Errores lógicos: se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo.

Depuración

La depuración es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores.
Cuando se ejecuta un programa se pueden producir tres tipos de errores:

- Errores de compilación: o *errores de sintaxis*, se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación.
- Errores de ejecución: se producen por instrucciones que la computadora puede comprender pero no ejecutar (división por cero, raíces cuadradas de números negativos).
- Errores lógicos: se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo.

Ejercicios

Realiza el análisis, el algoritmo y el diagrama de flujo de los siguiente problemas:

- Calcular la suma de dos números.
- Calcular el área de un triángulo.
- Calcular la hipotenusa de un triángulo.
- Identificar el mayor de dos números.
- Identificar el mayor de tres números.
- Imprimir los primeros diez números pares.

Introducción al lenguaje de programación

Estructura básica de un programa

```
/* Inclusion de librerias */

void main() /* cabecera de funcion */
{ /* inicio de la funcion main */

... /* Sentencias */

} /* fin de la funcion main */
```

Estructura básica de un programa para la clase

```
/* Inclusion de librerias */

void main() /* cabecera de funcion */
{ /* inicio de la funcion main */

/* Declaracion de variables */

/* Entrada de datos */

/* Procesamiento */

/* Impresion de resultados */

} /* fin de la funcion main */
```

Zonas de memoria

Variables

- Son objetos que pueden cambiar su valor durante la ejecución de un programa.
- En C una *variable* es una posición en memoria con nombre donde se almacena un valor de un cierto tipo de dato.

La *declaración* de una variable es una sentencia que proporciona información de la variable al compilador C. Su sintaxis es:

```
<tipo_variable> <nombre_variable> = <valor_inicial>;
```

Donde:

- *tipo_variable* es el nombre de un tipo de dato conocido por C.
- *nombre_variable* es un identificador (nombre) válido en C.
- *valor_inicial* es el valor de inicialización de la variable.

Zonas de memoria

Tipos de datos

Los tipos de datos básicos son:

- enteros
- flotantes
- caracteres

Constantes

Las *constantes* son datos que no cambian durante la ejecución de un programa.

```
#define NUEVALINEA \n
#define PI 3.141592
#define VALOR 54
```

Operadores

Operadores de asignación y expresión

Operador	Expresión	Explicación
+=	c += 7	c = c + 7
-=	d -= 4	d = d - 4
*=	e *= 5	e = e * 5
/=	f/= 3	f = f / 3
%=	g %= 9	g = g % 9

Operadores

Operadores aritméticos

Operador	Operación	Expresión algebraica	Expresión en C
+	Suma	f + 7	f + 7
-	Resta	p - c	p - c
*	Multiplicación	bm	b * m
/	División	$\frac{x}{y}$ ó x/y	x / y

Operadores

Operadores de relación

Operador	Ejemplo	Significado
==	x == y	x es igual que y
!=	x != y	x es diferente de y
>	x > y	x es mayor que y
<	x < y	x es menor que y
>=	x >= y	x es mayor o igual que y
<=	x <= y	x es menos o igual que y

Operadores

Operadores lógicos: AND lógico

expresión	expresión2	expresión1 && expresión2
0	0	0
0	diferente de cero	0
diferente de cero	0	0
diferente de cero	diferente de cero	1

Operadores

Operadores lógicos: OR lógico

expresión1	expresión2	expresión1 || expresión2
0	0	0
0	diferente de cero	1
diferente de cero	0	1
diferente de cero	diferente de cero	1

Operadores

Operadores lógicos: NOT lógico

expresión	!expresión
0	1
diferente de cero	0

Operadores

Operadores de incremento y decremento

Operador	Ejemplo	Explicación
++	++a	Incrementar a en 1 y después utiliza el nuevo valor de a en la expresión en la que reside.
++	a++	Utiliza el valor actual de a en la expresión en la que reside, y después la incrementa en 1.
−−	−−b	Decrementar b en 1 y después utiliza el nuevo valor de b en la expresión en la que reside.
−−	b−−	Utiliza el valor actual de b en la expresión en la que reside, y después decrementa b en 1.

Operadores

Jerarquía de operadores

Jerarquía	Operadores	Asociatividad	Tipo
Mayor	$++ \; -- \; + \; - \; !$	derecha a izquierda	unario
|	* / %	izquierda a derecha	multiplicativo
|	+ −	izquierda a derecha	aditivo
|	$< \; <= \; > \; >=$	izquierda a derecha	de relación
|	== !=	izquierda a derecha	de relación
|	&&	izquierda a derecha	AND lógico
|	||	izquierda a derecha	OR lógico
Menor	$=, \; + =, \; - =, \; * =, \; / =, \; \% =$	derecha a izquierda	de asignación

Instrucción de salida

La instrucción de salida utilizada en C se denomina **printf**. Su sintaxis es:

printf(“texto a imprimir”);
printf(“texto a imprimir %formato_de_impresión”,variable_a_imprimir);

Formato de salida con printf

Especificadores de conversión entera para *printf*

Especificador	Descripción
%d	Despliega un entero decimal con signo.
%i	Despliega un entero decimal con signo. [Nota: los especificadores i y d son diferentes cuando se utilizan con **scanf**.]
%o	Despliega un entero octal sin signo.
%u	Despliega un entero decimal sin signo.
%x ó **%X**	Despliega un entero hexadecimal sin signo. **X** provoca que se desplieguen los dígitos de **0** a **9** y las letras de **A** a **F**, y **x** provoca que se desplieguen los dígitos de **0** a **9** y las letras de **a** a **f**.
h ó **l** (letra l)	Se coloca antes de cualquier especificador de conversión entera para indicar que se despliega un entero corto o largo, respectivamente. Las letras **h** y **l** son llamadas con más precisión *modificadores de longitud*.

Formato de salida con printf

Especificadores de conversión de punto flotante para *printf*

Especificador	Descripción
%e ó **%E**	Despliega un valor de punto flotante con notación exponencial.
%f	Despliega un valor de punto flotante con notación de punto fijo.
%g ó **%G**	Despliega un valor de punto flotante con el formato de punto flotante **f**, o con el formato exponencial **e** (o **E**) basado en la magnitud del valor.
L	Se coloca antes del especificador de conversión para indicar que se desplegará un valor de punto flotante **long double**

Formato de salida con printf

Especificadores de conversión de caracteres y cadenas para *printf*

Especificador	Descripción
%c	Despliega caracteres individuales.
%s	Despliega cadenas de caracteres.

Formato de salida con printf

Otros especificadores de conversión para *printf*

Especificador	Descripción
%p	Despliega un valor apuntador de manera definida por la implementación.
%n	Almacena el número de caracteres ya desplegados en la instrucción **printf** actual. Proporciona un apuntador a un entero como el argumento correspondiente. No despliega valor alguno.
% %	Despliega el caracter de porcentaje.

Secuencias de escape

Secuencia de escape	Descripción
\a (alerta o campana)	Provoca una alerta sonora (campana) o una alerta visual.
\\(diagonal invertida)	Despliega el caracter de diagonal invertida (\).
\’ (comilla sencilla)	Despliega el caracter de comilla sencilla (’).
\” (comilla doble)	Despliega el caracter de comilla doble (”).
\? (interrogación)	Despliega el caracter de interrogación (?).
\n (nueva línea)	Mueve el cursor al inicio de la siguiente línea.

Secuencias de escape (Continuación)

Secuencia de escape	Descripción
\t (tabulador horizontal)	Mueve el cursor a la siguiente posición del tabulador.
\b (retroceso)	Mueve el cursor una posición hacia atrás en la línea actual.
\f (nueva página o avance de página)	Mueve el cursor al inicio de la siguiente página lógica.
\r (retorno de carro)	Mueve el cursor al principio de la línea actual.
\v (tabulador vertical)	Mueve el cursor a la siguiente posición del tabulador vertical.

Banderas de la cadena de control de formato

Bandera	**Descripción**
\– (signo menos)	Justifica la salida a la izquierda dentro del campo especificado.
\+ (signo más)	Despliega el signo más que precede a los valores positivos, y un signo menos que precede a los valores negativos.
espacio	Imprime un espacio antes de un valor positivo no impreso con la bandera +.

Instrucción de entrada

La instrucción de entrada utilizada en C se denomina **scanf**. Su sintaxis es:
scanf("especificador_de_conversión",&nombre_variable);
Donde:

- **especificador_de_conversión** describe el formato de los datos de entrada.
- **&** asigna los datos, en el formato especificado, a la variable especificada.
- **nombre_variable** variable a la cual se le asigna los datos de entrada.

Formato de entrada con scanf

Especificadores de conversión de enteros para *scanf*

Especificador	Descripción
%d	Lee un entero decimal con signo (el signo es opcional). El argumento correspondiente es un apuntador a un entero.
%i	Lee un entero decimal, octal, o hexadecimal con signo (opcional). El argumento correspondiente es un apuntador a un entero.
%o	Lee un entero octal. El argumento correspondiente es un apuntador a un entero sin signo.

Formato de entrada con scanf (Continuación)

Especificadores de conversión de enteros para *scanf*

Especificador	Descripción
%u	Lee un entero decimal sin signo. El argumento correspondiente es un apuntador a un entero sin signo.
%x o %X	Lee un entero hexadecimal. El argumento correspondiente es un apuntador a un entero sin signo.
h ó l	Se coloca antes de cualquier especificador de conversión, para indicar que se introducirá un entero corto o largo, respectivamente.

Formato de entrada con scanf

Especificadores de conversión de números de punto flotante para *scanf*

Especificador	Descripción
%e, %E, %f, %g ó %G	Lee un valor de punto flotante. El argumento correspondiente es un apuntador a un valor de punto flotante.
l ó L	Se coloca antes de cualquier especificador de conversión para indicar que se introducirá un valor **double** o **long double**. El argumento correspondiente es un apuntador a una varible **double** o **long double**.

Formato de entrada con scanf

Especificadores de conversión de cadenas y caracteres para *scanf*

Especificador	Descripción
%c	Lee un caracter. El argumento correspondiente es un apuntador a **char**; no agrega el caracter nulo ('\0').
%s	Lee una cadena. El argumento correspondiente es un apuntador a un arreglo de tipo **char** que sea lo suficientemente grande para almacenar la cadena y el caracter nulo ('\0'), el cual se agrega automáticamente.

Estructuras de selección

Estructuras de selección

Regulan el flujo de ejecución de un programa, permiten combinar instrucciones o sentencias individuales en una simple unidad lógica con un punto de entrada y un punto de salida.

L Joyanes Aguilar (2014). Programacion En C/C++ Java Y Uml (2da ed.). México, D.F.: McGrawHill.

Estructuras de selección

Las estructuras algorítmicas selectivas que se utilizan para la toma de decisiones lógicas las podemos clasificar de la siguiente forma:

- **SI - ENTONCES:** selección sencilla.
- **SI - ENTONCES / SINO:** selección doble.
- **SI MULTIPLE:** selección múltiple.

Selección sencilla

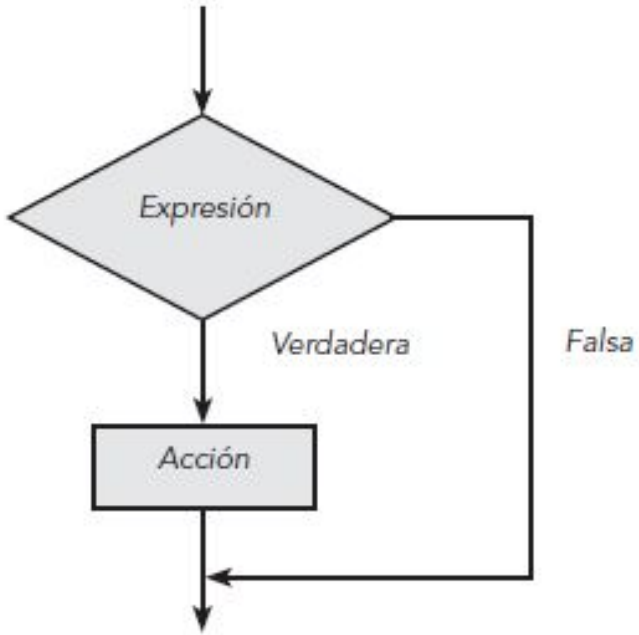
En C, la estructura de control de selección principal es una sentencia **if**. El formato tiene la sintaxis siguiente:

```
if(expresion)
{
    /*Acción*/
}
```

expresion: Expresión lógica que determina si la acción se ha de ejecutar.
Acción: Se ejecuta si la expresión lógica es verdadera.

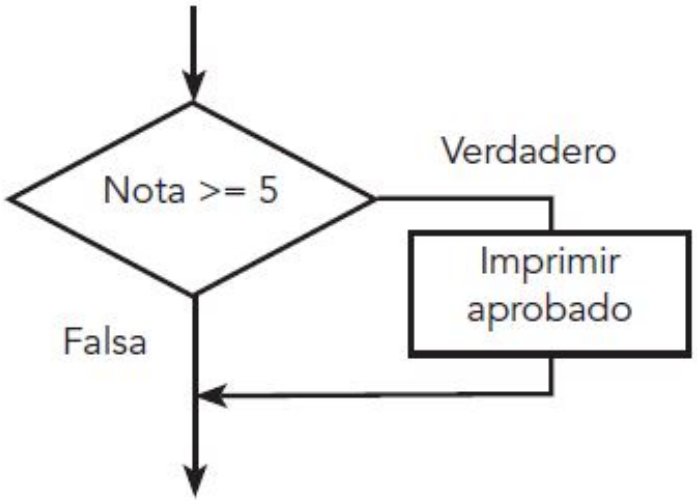
Selección sencilla

Diagrama de flujo de una sentencia básica *if*



Selección sencilla

Ejemplo: Representar la superación de un examen (Nota >= 5, Aprobado).



Selección sencilla

```c
#include <stdio.h>

void main()
{
    float nota;
    printf("Introduzca la nota obtenida (0-10): ");
    scanf("%f", &nota);
    /* compara nota con 5 */
    if (nota >= 5)
        printf("Aprobado");
}
```

Selección doble

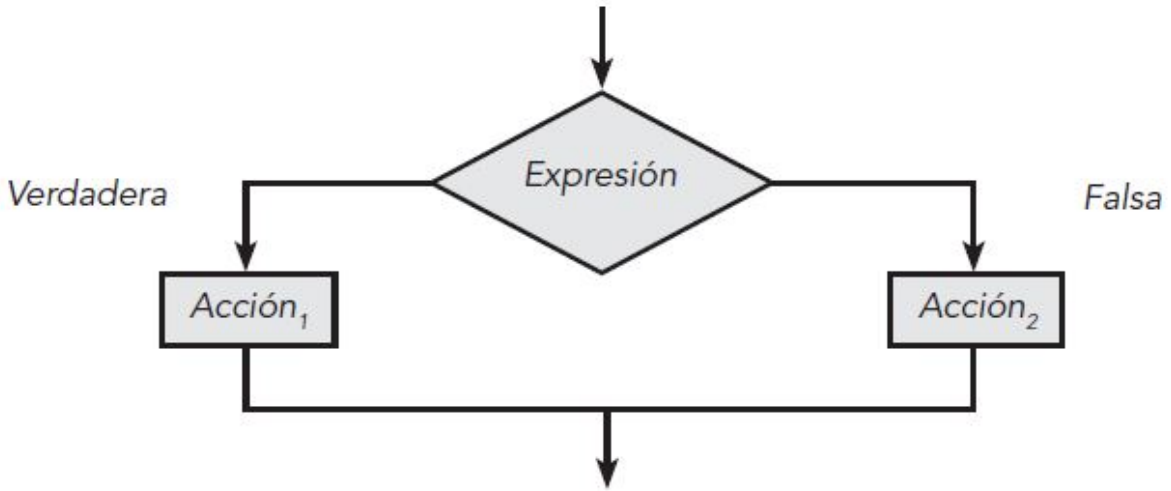
En este formato la sentencia **if** tiene la siguiente sintaxis:

if (Expresión)	Accion$_1$	*else* Accion$_2$
Expresión lógica que determina la acción a ejecutar	Acción que se realiza si la expresión lógica es verdadera.	Acción que se ejecuta si la expresión lógica es falsa.

Selección doble

Acción$_1$ y Acción$_2$, son individualmente, o bien una única instrucción que termina en un punto y coma (;) o un grupo de instrucciones encerradas entre llaves.

Si *Expresión* es verdadera, se ejecuta *Acción*$_1$ y en caso contrario se ejecuta *Acción*$_2$



Selección doble

Ejemplo: Calcular el mayor de dos números leídos del teclado y visualizarlo en pantalla.

```c
#include <stdio.h>
int main()
{
    int valor1, valor2;
    printf("Ingrese el valor 1: ");
    scanf("%d", &valor1);
    printf("Ingrese el valor 2: ");
    scanf("%d", &valor2);
    if(valor1>valor2)
        printf("El mayor es: %d", valor1);
    else
        printf("El mayor es: %d", valor2);
    return 0;
}
```

Selección Multiple

La sentencia **switch** es una sentencia que se utiliza para seleccionar una de múltiples alternativas. Es especialmente útil cuando la selección se basa en el valor de una variable simple o de una expresión simple denominada ***expresión de control*** o ***selector***. El valor de esta expresión puede ser de tipo ***int*** o ***char***, pero no de tipo float ni double.

Selección Multiple

SINTAXIS

```
switch (selector)
{
    case etiqueta1 : sentencias1;
    case etiqueta2 : sentencias2;
    .
    .
    case etiquetan : sentenciasn;
    default: sentencias; /* opcional */
}
```

Selección Multiple

La expresión de control o **selector** se evalúa y se compara con cada una de las etiquetas de **case**. Cada etiqueta es un valor único, constante y cada etiqueta debe tener un valor diferente de los otros.

Si el valor de la expresión selector es igual a una de las etiquetas case, por ejemplo, etiqueta1, entonces la ejecución comenzará con la primera sentencia de la secuencia y continuará hasta que se encuentre el final de la sentencia de control switch, o hasta encontrar la sentencia **break**.

Típicamente después de cada bloque se sitúa la sentencia **break** como última sentencia del bloque. La sentencia **break** hace que siga la ejecución en la siguiente sentencia al switch.

Selección Multiple

SINTAXIS CON break

```
switch (selector)
{
    case etiqueta1 : sentencias1;
    break;
    case etiqueta2 : sentencias2;
    break;
    .
    .
    .
    case etiquetan :sentenciasn;
    break;
    default: sentenciasd; /* opcional */
}
```

Selección Multiple

Elección de tres opciones y un valor en forma predeterminada.

```
switch (opcion)
{
    case 0:
        printf("Cero!");
        break;
    case 1:
        printf("Uno!");
        break;
    case 2:
        printf("Dos!");
        break;
    default:
        printf("Fuera de rango");
}
```

Selección Multiple

Selección múltiple, tres etiquetas ejecutan la misma sentencia.

```
switch (opcion)
{
    case 0:
    case 1:
    case 2:
        printf("Menor de 3");
        break;
    case 3:
        printf("Igual a 3");
        break;
    default:
        printf("Mayor que 3");
}
```

Selección Multiple

Una sentencia **if** es anidada cuando la sentencia de la rama verdadera o la rama falsa es a su vez una sentencia if. Una sentencia **if** anidada se puede utilizar para implementar decisiones con varias alternativas o multialternativas.

SINTAXIS:

```
if(condición_1)
    sentencia_1
else if(condicion_2)
    sentencia_2
.
.
else if(condicion_n-1)
    sentencia_n-1
else
    sentencia_n
```

Selección Multiple

Comparación de las sentencias if-else-if y switch. Se necesita saber si un determinado carácter car es una vocal. Solución con if-else-if.

```
if((car=='a')||(car=='A'))
   printf("%c es una vocal\n",car)
         ;
else if((car=='e')||(car=='E'))
   printf("%c es vocal\n",car);
else if((car=='i')||(car=='I'))
   printf("%c es vocal\n",car);
else if((car=='o')||(car=='O'))
   printf("%c es vocal\n",car);
else if((car=='u')||(car=='U'))
   printf("%c es vocal\n",car);
else
   printf("%c no es vocal\n",car);
```

```
switch (car)
{
     case 'a': case 'A':
     case 'e': case 'E':
     case 'i': case 'I':
     case 'o': case 'O':
     case 'u': case 'U':
          printf("%c es una vocal\n",car);
          break;
     default:
        printf("%c no es una vocal\n",car);
}
```

Teoría de ciclos

Teoría de Ciclos

Definición

Un bucle (ciclo o lazo, loop en inglés) es cualquier construcción de programa que repite una sentencia o secuencia de sentencias un número de veces. La sentencia (o grupo de sentencias) que se repiten en un bloque se denomina cuerpo del bucle y cada repetición del cuerpo del bucle se llama iteración del bucle.

Teoría de Ciclos

Contadores

Es una variable cuyo valor se incrementa o decrementa en una cantidad constante cada vez que se produce un determinado suceso o acción en cada repetición; dicha variable controla o determina la cantidad de veces que se repite un proceso o dato.

Contadores

```
int contador 5 1; //variable con valor inicial de 1
contador++; o ++contador; o contador+=1;
```

Teoría de Ciclos

Acumuladores

Un acumulador realiza la misma función que un contador con la diferencia de que el incremento o decremento es variable. Es una variable que acumula sobre sí misma un conjunto de valores, para tener la acumulación de todos ellos en una sola variable. Almacena cantidades resultantes de operaciones sucesivas.

Acumuladores

```
int acumulador 50;
acumulador = acumulador + valor;
```

Teoría de Ciclos

Centinela

El centinela es una variable que inicia con un valor, luego dentro de un bucle este valor cambia, haciendo falsa la condición del ciclo y por lo tanto indicará el fin del ciclo (el usuario puede determinar cuándo hacerlo). La repetición controlada por centinela se considera como una repetición indefinida (se desconoce el número de repeticiones).

Teoría de ciclos

Ciclo While

Un ciclo **while** tiene una condición (expresión lógica) que controla la secuencia de repetición. La posición de esta condición es previo al cuerpo del ciclo, de modo que, cuando se ejecuta, se evalúa la condición antes de que se ejecute el cuerpo del bucle.

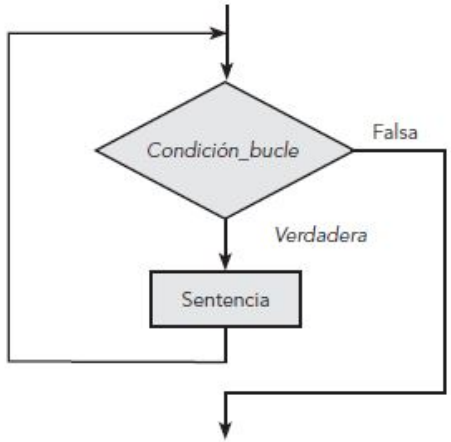
Tipos de Ciclos

Sintaxis

```
while(condicion)
    sentencia;
```

```
while(condicion)
{
    senctencia_1;
    .
    .
    sentencia_n;
}
```

Teoría de ciclos



El diagrama indica que la ejecución de la sentencia se repite mientras la condición permanece verdadera y termina cuando se hace falsa. También que la condición del ciclo se evalúa antes de ejecutar el cuerpo del ciclo, si esta condición es inicialmente falsa, el cuerpo del ciclo no se ejecutará. En otras palabras, el cuerpo de un bucle while se ejecutará cero o más veces.

Teoría de ciclos

CICLO DE MUESTRA CON *WHILE*

```c
#include<stdio.h>

int main()
{
    int contador = 0;
    while(contador<5)
        printf("contador: %d  \n", ++contador);
    printf("Terminado. Contador: %d\n", contador);
    return 0;
}
```

Teoría de ciclos

Ciclo Infinito

Si la variable de control no se actualiza el ciclo se ejecutará "siempre" (ciclo infinito). Un ciclo infinito (sin terminación) se produce cuando la condición del permanece y no se hace falsa en ninguna iteración.

```
/* bucle infinito */
contador = 1;
while (contador < 100)
{
    printf("%d \n", contador);
    contador--;
}
```

Tipos de ciclos

Ciclos controlados por contador

La repectición controlada por un contador se utiliza cuando se conoce el número de repeticiones antes de que un ciclo empiece a ejecutarse; es decir, cuando hay una repetición definida.

Tipos de ciclos

Ciclos controlados por centinela

La repetición controlada por un centinela se utiliza cuando no se conoce el número de repeticiones antes de que un ciclo empiece a ejecutarse; es decir, cuando hay una repetición indefinida.

Tipos de ciclos

Ejemplos

Por contador

El siguiente ciclo cuenta hasta 10

```
int x = 0;
while( x < 10 )
    printf("X: %d\n", x++);
```

Por centinela

Visualizar n asteriscos

```
contador = 0;
while( contador < n )
{
    printf(" * ");
    contador++;
}/*Fin del while*/
```

Tipos de ciclos

Ciclo For

La sentencia **for** es un método para ejecutar un bloque de sentencias un número fijo de veces. El *ciclo* ***for*** se diferencia del *ciclo* ***while*** en que las operaciones de control del ciclo se sitúan en un solo sitio: la cabecera de la sentencia.

Tipos de ciclos

Sintaxis

```
for(Inicialización; CondiciónIteración;Incremento)
    sentencias
```

- Inicialización: Inicializa la variable de control del ciclo.
- CondiciónIteración: Expesión lógica que determina se las sentencias se han de ejecutar (mientras sea verdadera)
- Incremento: Incrementa o decrementa la variable de control de bucle.
- sentencias: Sentencias a ejecutar en cada iteración del bucle

Tipos de ciclos

Sintaxis

```
for(Inicialización; CondiciónIteración;Incremento)
    sentencias
```

- Inicialización: Inicializa la variable de control del ciclo.
- CondicióIteración: Expesión lógica que determina se las sentencias se han de ejecutar (mientras sea verdadera)
- Incremento: Incrementa o decrementa la variable de control de bucle.
- sentencias: Sentencias a ejecutar en cada iteración del bucle

Tipos de ciclos

Sintaxis

```
for(Inicialización; CondiciónIteración;Incremento)
    sentencias
```

- Inicialización: Inicializa la variable de control del ciclo.
- CondicióIteración: Expesión lógica que determina se las sentencias se han de ejecutar (mientras sea verdadera)
- Incremento: Incrementa o decrementa la variable de control de bucle.
- sentencias: Sentencias a ejecutar en cada iteración del bucle

Tipos de ciclos

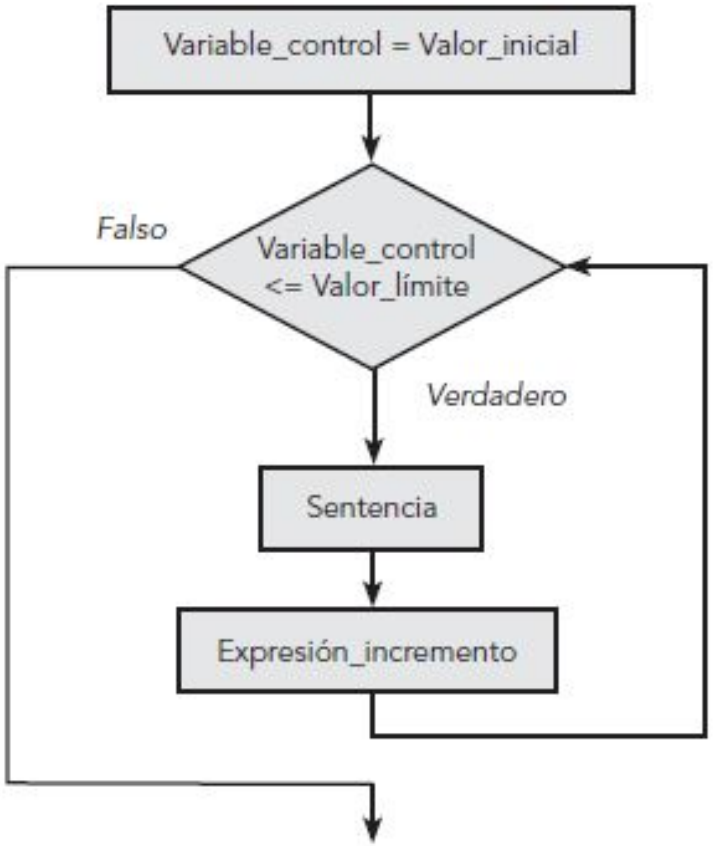
Sintaxis

```
for(Inicialización; CondiciónIteración;Incremento)
    sentencias
```

- Inicialización: Inicializa la variable de control del ciclo.
- CondicióIteración: Expesión lógica que determina se las sentencias se han de ejecutar (mientras sea verdadera)
- Incremento: Incrementa o decrementa la variable de control de bucle.
- sentencias: Sentencias a ejecutar en cada iteración del bucle

Tipos de ciclos

Diagrama de Flujo del ciclo *For*



Tipos de ciclos

Ejemplo

```
#include<stdio.h>

int main()
{
    int n, suma = 0;
    for(n=2; n<=20; n+=2)
        suma += n;
    printf("La suma de los 10 primeros numeros pares es:
        %d", suma);
    return 0;
}
```

Tipos de ciclos

Ciclo do-while

La sentencia ***do-while*** se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener la seguridad de que una determinada acción se ejecutará una o varias veces, pero al menos una vez.

Tipos de ciclos

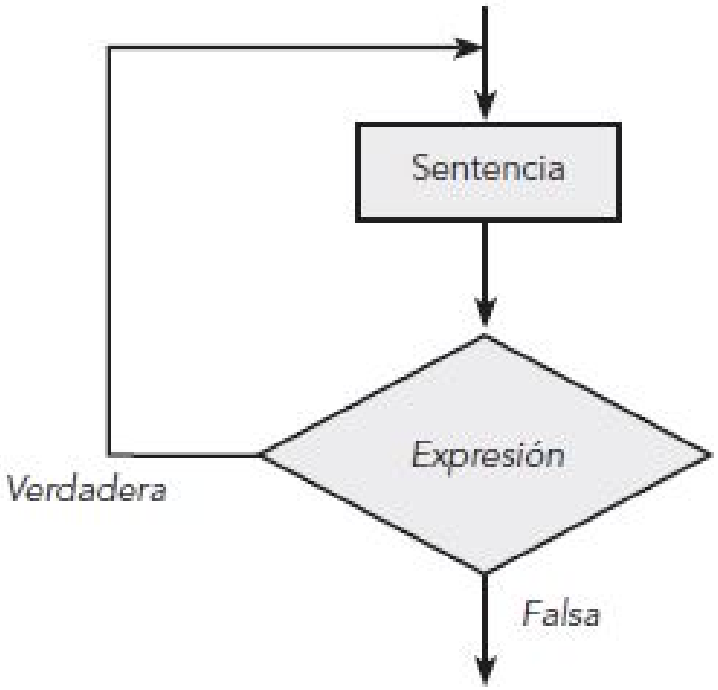
SIntaxis

```
do
    sentencia
while(expresion);
```

```
do
{
    sentencias;
}while(expresion);
```

Tipos de ciclos

Sintaxis



Después de cada ejecución de sentencia se evalúa expresión: si es falsa, se termina el ciclo y se ejecuta la siguiente sentencia; si es verdadera, se repite el cuerpo del ciclo (la sentencia).

Tipos de ciclos

Ejemplo

```c
#include<stdio.h>

int main()
{
    int i=1;
    do
    {
        printf("%d\n", i);
        i++
    }while(i<11);
    return 0;
}
```

Cadenas de caracteres

Cadena de caracteres

En el lenguaje C no existe el tipo de dato cadena (string) como en otros lenguajes de programación, por lo que se utiliza un arreglo (lista finita de n cantidad de elementos del mismo tipo) de caracteres, para poder almacenar una cadena:

```
char cad[] = "Lenguaje";
```

Una cadena de caracteres es un arreglo de caracteres que contiene al final el carácter nulo (\0); por esta razón es necesario que al declarar los arreglos éstos sean de un carácter más que la cadena más grande. El compilador inserta automáticamente un carácter nulo al final de la cadena, de modo que la secuencia real sería:

```
char cad[9] = "Lenguaje";
```

Cadena de caracteres

Lectura

Una opción para almacenar una cadena de caracteres es el uso de la palabra reservada **scanf(variable)** pero, si queremos almacenar una cadena con espacios en blanco no lo podemos hacer con ella, sino que debemos utilizar la palabra reservada **gets**, que se encuentra dentro de la librería string.h; **gets** sólo se utiliza para leer cadenas de caracteres y scanf para leer cualquier tipo de variable, de preferencia de tipo numérico.

Cadena de caracteres

Ejemplo:

```c
#include<string.h>

int main(void)
{
    char frase[100];
    printf("Ingrese una frase: ");
    gets(frase);
    printf("Ingresaste: %s", frase);
    return 0;
}
```

Cadena de caracteres

Escritura

La función **puts()**, escribe una cadena de caracteres de salida y remplaza el caracter nulo de terminación de la cadena (\0) por el carácter de nueva línea (\n).

Cadena de caracteres

Asignación de Cadenas

C soporta dos métodos para asignar cadenas. Uno de ellos, ya visto anteriormente, cuando se inicializaban las variables de cadena. La sintaxis utilizada:

```
char cadena[longitudCadena] = "ConstanteCadena";
```

Cadena de caracteres

Asignación de Cadenas

El segundo método para asignación de una cadena a otra es utilizar la función strcpy( ). La función copia los caracteres de la cadena fuente a la cadena destino. La cadena destino debe tener espacio suficiente para contener toda la cadena fuente. El prototipo de la función:

```
char* strcpy(char* destino, const char* fuente);
```

Una vez definido el arreglo de caracteres, se le asigna una cadena constante:

```
char nombre[41];
strcpy(nombre, "Cadena a copiar");
```

Cadena de caracteres

Comparación de Cadenas

La biblioteca string.h proporciona un conjunto de funciones que comparan cadenas. Estas funciones comparan los caracteres de dos cadenas utilizando el valor ASCII de cada carácter. La funcion es strcmp( ).

```
int strcmp(const char* cad1, const char* cad2);
```

La función compara las cadenas cad1 y cad2. El resultado entero es:

- $<$ 0 si cad1 es menor que cad2.
- $=$ 0 si cad1 es igual a cad2.

Cadena de caracteres

Ejemplo de Comparación de Cadenas

```
char cad1[ ] = "Microsoft C";
char cad2[ ] = "Microsoft Visual C"
int i;
i = strcmp(cad1, cad2); /*i, toma un valor negativo */
strcmp("Waterloo", "Windows") /*< 0 {Devuelve un valor negativo}*/
strcmp("Mortimer", "Mortim") /*> 0 {Devuelve un valor positivo}*/
strcmp("Jertru", "Jertru") /*= 0 {Devuelve cero}*/
```

Arreglos unidimencionales

Arreglos Unidimencionales

Definición

Un arreglo es un tipo de dato estructurado que almacena en una sola variable un conjunto limitado de elementos del mismo tipo. El nombre del arreglo apunta a la dirección del primer elemento del arreglo. Los datos se llaman elementos del arreglo y su posición se numera consecutivamente: 1, 2, 3...n. Un arreglo inicia en la posición cero, por lo tanto el i-ésimo elemento está en la posición i-1, es decir si el arreglo llamado **a** tiene ***n*** elementos, sus nombres son a[0], a[1], ..., a[n-1].

Arreglos Unidimencionales

Para acceder a un elemento específico de un arreglo se usa un índice o subíndice. Un arreglo se caracteriza por:

1. Ser una lista de un número finito de n elementos del mismo tipo.
2. Almacenar los elementos del arreglo en memoria contigua.
3. Tener un único nombre de variable que representa a todos los elementos y éstos se diferencian por un índice o subíndice.
4. Acceder de manera directa o aleatoria a los elementos individuales del arreglo, por el nombre del arreglo y el índice o subíndice.

Arreglos Unidimencionales

Formato para declarar un arreglo unidimencional.

```
tipoDato identifArreglo[tamArreglo];
```

Donde:

- **tipoDato** se refiere al tipo de dato de cada elemento del arreglo; puede ser entero, real, carácter, etcétera.
- **identifArreglo** es el nombre que representa a todo el arreglo.
- **tamArreglo** es la cantidad de elementos que contiene el arreglo.

Arreglos Unidimencionales

Formato para declarar un arreglo unidimencional.

```
tipoDato identifArreglo[tamArreglo];
```

Donde:

- **tipoDato** se refiere al tipo de dato de cada elemento del arreglo; puede ser entero, real, carácter, etcétera.
- **identifArreglo** es el nombre que representa a todo el arreglo.
- **tamArreglo** es la cantidad de elementos que contiene el arreglo.

Arreglos Unidimencionales

Formato para declarar un arreglo unidimencional.

```
tipoDato identifArreglo[tamArreglo];
```

Donde:

- **tipoDato** se refiere al tipo de dato de cada elemento del arreglo; puede ser entero, real, carácter, etcétera.
- **identifArreglo** es el nombre que representa a todo el arreglo.
- **tamArreglo** es la cantidad de elementos que contiene el arreglo.

Arreglos Unidimencionales

Formato para declarar un arreglo unidimencional.

```
tipoDato identifArreglo[tamArreglo];
```

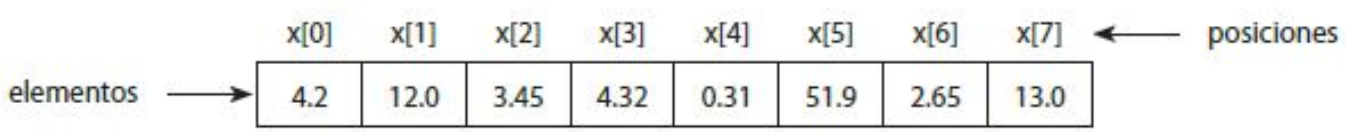
Donde:

- **tipoDato** se refiere al tipo de dato de cada elemento del arreglo; puede ser entero, real, carácter, etcétera.
- **identifArreglo** es el nombre que representa a todo el arreglo.
- **tamArreglo** es la cantidad de elementos que contiene el arreglo.

Arreglos Unidimencionales

Ejemplo

```
float x[8];
```



Este arreglo contiene ocho elementos almacenados entre la posición (0-7). Para referirnos a un elemento en particular dentro del arreglo, especificamos el nombre del arreglo y el número de posición donde se encuentra ubicado. La posición del arreglo va entre corchetes (“[ ]”).

Arreglos Unidimencionales

Si la instrucción fuera imprimir **x[4]** se mostrará el valor de 0.31

```
printf("%f", x[4]);
```

Para almacenar la suma de los valores contenidos en los primeros tres elementos del arreglo x, escribiríamos:

```
a = x[0] + x[1] + x[2];
printf("%f", a);
```

Para dividir el valor del séptimo elemento del arreglo x entre 2 y asignar el resultado a la variable c escribiríamos:

```
c = x[6]/2;
```

Arreglos Unidimencionales

Inicialización

En el momento de declarar el arreglo, se especifican los valores.

```
/*tipoDato identif[tamArreglo]={valores};*/
int lista[5] = {10,17,8,4,9};
```

Para llenar un arreglo completo se utiliza generalmente el ciclo desde (**for**) facilitando con la variable de control el incremento de la ***i***, donde la ***i*** representa el subíndice.

```
for(i=0;i<5;i++)
    printf("%d", lista[i]);
```

Arreglos Unidimencionales

Modificación

Para modificar los elementos de un vector en cualquier momento, sólo es necesario especificar el nombre del arreglo unidimensional, la posición y el nuevo valor. Enseguida se muestra la sintaxis a seguir:

```
/*tipoDato identArr[pos]=valor;*/
int b[3] = 18;
```

Donde **valor** es un dato, el resultado de alguna operación lógica o aritmética, etc.

Arreglos bidimencionales

Arreglos Bidimencionales

Definición

Un arreglo bidimensional (tabla, matriz) es un conjunto de **n** elementos del mismo tipo almacenados en una matriz o tabla. A diferencia de los arreglos unidimensionales que sólo requieren de un subíndice, los arreglos bidimensionales para acceder a cada elemento del arreglo requieren de dos índices o subíndices declarados en dos pares de corchetes, donde el primer corchete se refiere al tamaño de filas y el segundo al tamaño de columnas.

Arreglos Bidimencionales

Declaración de un arreglo bidimencional

```
tipoDato identArr[tamFila][tamCol];
```

Donde:

- tipoDato es el tipo de dato de todo el arreglo.
- identArr es el nombre del arreglo.
- tamFila es el total de filas.
- tamCol es el total de columnas.

Arreglos Bidimencionales

Declaración de un arreglo bidimencional

```
tipoDato identArr[tamFila][tamCol];
```

Donde:

- tipoDato es el tipo de dato de todo el arreglo.
- identArr es el nombre del arreglo.
- tamFila es el total de filas.
- tamCol es el total de columnas.

Arreglos Bidimencionales

Declaración de un arreglo bidimencional

```
tipoDato identArr[tamFila][tamCol];
```

Donde:

- tipoDato es el tipo de dato de todo el arreglo.
- identArr es el nombre del arreglo.
- tamFila es el total de filas.
- tamCol es el total de columnas.

Arreglos Bidimencionales

Declaración de un arreglo bidimencional

```
tipoDato identArr[tamFila][tamCol];
```

Donde:

- tipoDato es el tipo de dato de todo el arreglo.
- identArr es el nombre del arreglo.
- tamFila es el total de filas.
- tamCol es el total de columnas.

Arreglos Bidimencionales

Ejemplo:
Declaración de un arreglo de 3 filas y 5 columnas

```
int b[3][5];
```

b	Col 0	Col 1	Col 2	Col 3	Col 4
Fila 0	b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]
Fila 1	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]
Fila 2	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

El número de elementos de una matriz será tamFila 3 tamCol.
En el ejemplo anterior son 3 x 5 = 15 celdas de tipo entero.

Arreglos Bidimencionales

Inicialización

En el momento de declarar el arreglo, se especifican los valores:

```
/*tipoDato identif[fil][col] = {valores};*/
int a[3][3] = {1,2,3,4,5,6,7,8,9};
```

a	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Arreglos Bidimencionales

Lectura e Impresión

Para la lectura o escritura se requieren de dos ciclos anidados (para ubicar la fila y la columna)en cada celda de la tabla o matriz. El siguiente segmento de programa muestra cómo se pueden almacenar datos en una matriz mat de 3 filas y 4 columnas, se utiliza la instrucción leer (scanf) para guardar o leer los datos:

```
for(i=0;i<3;i++)
     for(j=0;j<4;j++)
          scanf("%d", &mat[i][j]);
```

Arreglos Bidimencionales

El siguiente segmento de programa muestra cómo se pueden imprimir los datos almacenados en una matriz mat de 3 filas y 4 columnas. Se utiliza la instrucción imprimir (printf) para escribir o mostrar el resultado:

```
for(i=0;i<3;i++)
    for(j=0;j<4;j++)
        printf("%d", mat[i][j]);
```

Arreglos Bidimencionales

Modificación de un elemento de un arreglo Bidimencional

Los elementos de un arreglo bidimencional se pueden modificar en cualquier momento, sólo es necesario especificar el nombre del arreglo bidimensional, la posición tanto de la fila como de la columna y el nuevo valor. En seguida se muestra la sintaxis a seguir:

```
identArr [fil][col] = valor;
```

Donde valor es un dato o el resultado de alguna operación lógica o aritmética, etc.

Definición de Funcion

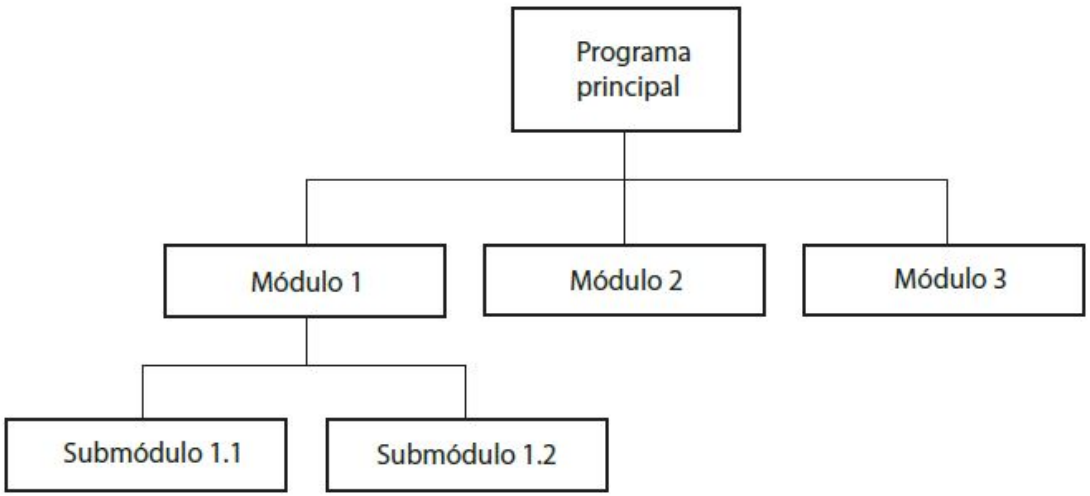
Definición de Funcion

Un problema difícil es más sencillo al dividirlo en pequeñas partes y tratar de buscar la solución de cada una de ellas y así resolver todo el problema general, la mejor forma de elaborar y dar mantenimiento a un programa complejo es construirlo a partir de bloques menores o módulos.

Dichos módulos se escriben solamente una vez, pero pueden ser llamados en diferentes puntos del programa principal o de cualquier otro módulo.

Definición de Funcion

En lenguaje C a cada módulo o subprograma se le conoce como función; en este lenguaje se trabaja a base de funciones y, de manera general, los programas se elaboran combinando funciones que el programador escribe y funciones “predefinidas” disponibles en la biblioteca estándar de C.



Definición de Funcion

Ventajas de la programación modular

- Facilita el diseño descendente *(proceso mediante el cual un problema se descompone en un serie de niveles o pasos sucesivos)*.
- Se simplifica un algoritmo complejo.
- Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y disminuir el tiempo de elaboración del algoritmo.
- La depuración se lleva a cabo en cada módulo.
- El mantenimiento es más sencillo.
- Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Definición de Funcion

Ventajas de la programación modular

- Facilita el diseño descendente *(proceso mediante el cual un problema se descompone en un serie de niveles o pasos sucesivos)*.
- Se simplifica un algoritmo complejo.
- Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y disminuir el tiempo de elaboración del algoritmo.
- La depuración se lleva a cabo en cada módulo.
- El mantenimiento es más sencillo.
- Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Definición de Funcion

Ventajas de la programación modular

- Facilita el diseño descendente *(proceso mediante el cual un problema se descompone en un serie de niveles o pasos sucesivos)*.
- Se simplifica un algoritmo complejo.
- Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y disminuir el tiempo de elaboración del algoritmo.
- La depuración se lleva a cabo en cada módulo.
- El mantenimiento es más sencillo.
- Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Definición de Funcion

Ventajas de la programación modular

- Facilita el diseño descendente *(proceso mediante el cual un problema se descompone en un serie de niveles o pasos sucesivos)*.
- Se simplifica un algoritmo complejo.
- Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y disminuir el tiempo de elaboración del algoritmo.
- La depuración se lleva a cabo en cada módulo.
- El mantenimiento es más sencillo.
- Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Definición de Funcion

Ventajas de la programación modular

- Facilita el diseño descendente *(proceso mediante el cual un problema se descompone en un serie de niveles o pasos sucesivos)*.
- Se simplifica un algoritmo complejo.
- Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y disminuir el tiempo de elaboración del algoritmo.
- La depuración se lleva a cabo en cada módulo.
- El mantenimiento es más sencillo.
- Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Definición de Funcion

Ventajas de la programación modular

- Facilita el diseño descendente *(proceso mediante el cual un problema se descompone en un serie de niveles o pasos sucesivos)*.
- Se simplifica un algoritmo complejo.
- Cada módulo se puede elaborar de manera independiente, lo que permite trabajar simultáneamente a varios programadores y disminuir el tiempo de elaboración del algoritmo.
- La depuración se lleva a cabo en cada módulo.
- El mantenimiento es más sencillo.
- Creación de bibliotecas con módulos específicos (se pueden utilizar en otros programas).

Definición de Funcion

Función

Es un subprograma que realiza una tarea específica que puede o no recibir valores (parámetros). En C podemos devolver cualquier tipo de datos escalares tipo numérico y el tipo carácter o en su caso regresar un valor nulo que llamaremos nada o ninguno). El uso de funciones es una práctica común y recomendable ya que permite dividir el código, simplificando así el desarrollo y la depuración del mismo.

Prototipos, llamada y cuerpo de la función

Prototipos, llamada y cuerpo de la función

Prototipo

Un prototipo es el encabezado de una función, es decir la primera línea de la función. La ventaja de utilizar prototipos es que las funciones pueden estar en cualquier lugar y en cualquier orden y pueden ser llamadas desde cualquier punto del programa principal o de otra función.

Orden

1. Declaración de los prototipos de cada función.
2. Cuerpo de la función principal.
3. Al final el cuerpo de cada función.

Prototipos, llamada y cuerpo de la función

Prototipo

Un prototipo es el encabezado de una función, es decir la primera línea de la función. La ventaja de utilizar prototipos es que las funciones pueden estar en cualquier lugar y en cualquier orden y pueden ser llamadas desde cualquier punto del programa principal o de otra función.

Orden

1. Declaración de los prototipos de cada función.
2. Cuerpo de la función principal.
3. Al final el cuerpo de cada función.

Prototipos, llamada y cuerpo de la función

Prototipo

Un prototipo es el encabezado de una función, es decir la primera línea de la función. La ventaja de utilizar prototipos es que las funciones pueden estar en cualquier lugar y en cualquier orden y pueden ser llamadas desde cualquier punto del programa principal o de otra función.

Orden

1. Declaración de los prototipos de cada función.
2. Cuerpo de la función principal.
3. Al final el cuerpo de cada función.

Prototipos, llamada y cuerpo de la función

EJEMPLO

```
#librearias
#Constantes
funcionA();/*Prototípo de la función A*/
funcionB();/*Prototípo de la función B*/
int main(){
    variables
    Cuerpo del programa principal
    ............
    funcionaA();
    funcionaB();
    .......
    return 0;
}
funcionA(){
    variables locales de función
    instrucciones
}
funcionB(){
    variables locales de función
    instrucciones
}
```

Funciones Sencillas

Funciones Sencillas

Son aquellas que no reciben parámetros o valores, ya que éstos se solicitan dentro de la función, luego se realizan las instrucciones (cálculos u operaciones) y normalmente se imprime el resultado.
Ejemplo:

```
void nombreFuncion() {
    Declaracion de variables;
    Cuerpo de la funcion;
    .
    .
}
```

Funciones Sencillas

Llamadas a funciones sin paso de parametros

```
printf("%d", nombreFuncion());

variable = nombreFuncion();

if(nombreFuncion() > expresion)
```

Funciones con parámetros por valor y que regresan valor

Funciones con parámetros por valor y que regresan valor

Estas funciones son las más utilizadas en la programación ya que pueden recibir uno o más valores llamados parámetros y regresan un solo valor de tipo entero, real o carácter. Si deseas regresar un arreglo de carácter es necesario hacerlo desde los parámetros. Los parámetros o valores son enviados del programa principal o de otra función. Dentro de la función se realizan solamente las instrucciones (cálculos u operaciones). Es importante revisar que el tipo de dato que regresará la función sea del mismo tipo que el valor declarado en el encabezado de la misma.

Funciones con parámetros por valor y que regresan valor

Parámetros de una Función

También son llamados argumentos y se corresponden con una serie de valores que se especifican en la llamada a la función, o en la declaración de la misma, de los que depende el resultado de la función; dichos valores nos permiten la comunicación entre dos funciones.

Funciones con parámetros por valor y que regresan valor

Paso de parámetros en una Función

En C todos los parámetros se pasan “por valor”, es decir, en cada llamada a la función se genera una copia de los valores de los parámetros actuales, que se almacenan en variables temporales mientras dure la ejecución de la función. Sin embargo, cuando sea preciso es posible hacer que una función modifique el valor de una variable que se pase como parámetro actual en la llamada a la función. Para ello, lo que se debe proporcionar a la función no es el valor de la variable, sino su dirección, lo cual se realiza mediante un puntero que señale a esa dirección; a estos parámetros los llamamos por variable o por referencia; en este tipo de parámetros los datos salen modificados.

Funciones con parámetros por valor y que regresan valor

```
int suma(int numA, int numB);
int main(void){
    int a;
    int b = 3;
    a = suma(b,6);
    printf("%d", a);
    return 0;
}

int suma(int numA, int numB){
    return numA + numB;
}
```

Funciones con parámetros por valor y que regresan valor

Paso de parámetros en una funciones con arreglos unidimencionales y bidimencionales

- Unidimencional:
  tipoDevuelto nombreFunción(tipo nombreVector[])
- Bidimencional:
  tipoDevuelto nombreFunción(tipo nombreMatriz[][tam])

Gracias