

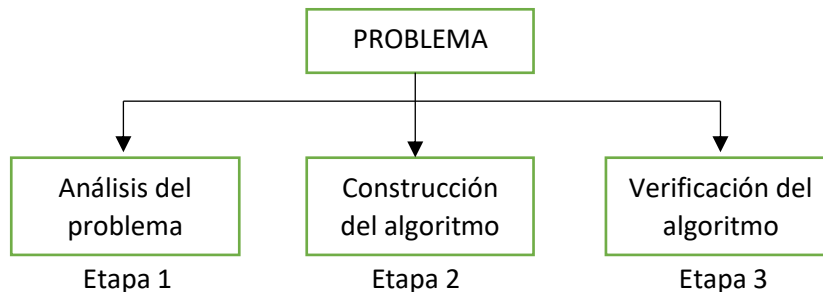
Capítulo 1 - Algoritmos, diagramas de flujo y programas en C

1.1 Problemas y algoritmos

- Los humanos efectuamos cotidianamente una serie de pasos, procedimientos o acciones, que nos permiten alcanzar algún resultado o resolver algún problema.
- Esta serie de pasos, procedimientos o acciones, comenzamos a aplicarla desde que empieza el día, cuando, por ejemplo, decidimos bañarnos.
- Posteriormente cuando tenemos que ingerir alimentos, también seguimos una serie de pasos que nos permite alcanzar un resultado específico: *tomar el desayuno*.
- La historia se repite innumerables veces durante el día. En realidad, todo el tiempo estamos aplicando **algoritmos para resolver problemas**.

○ Formalmente definimos un algoritmo como un conjunto de pasos, procedimientos o acciones que nos permiten alcanzar un resultado o resolver un problema.

- Muchas veces aplicamos el algoritmo de manera inadvertida, inconsciente o automática. Esto ocurre generalmente cuando el problema al que nos enfrentamos, lo hemos resuelto con anterioridad un gran número de veces.
- Supongamos que tenemos que abrir una puerta. Lo hemos hecho tantas veces que difícilmente nos tomamos la molestia de enumerar los pasos para alcanzar este objetivo. Lo hacemos de manera automática. Lo mismo ocurre cuando nos subimos a un automóvil, lustramos nuestros zapatos, hablamos por teléfono, nos vestimos, cambiamos la llanta de un automóvil o simplemente cuando tomamos un vaso con agua.
- Podemos observar las etapas que debemos seguir para solucionar algún problema.



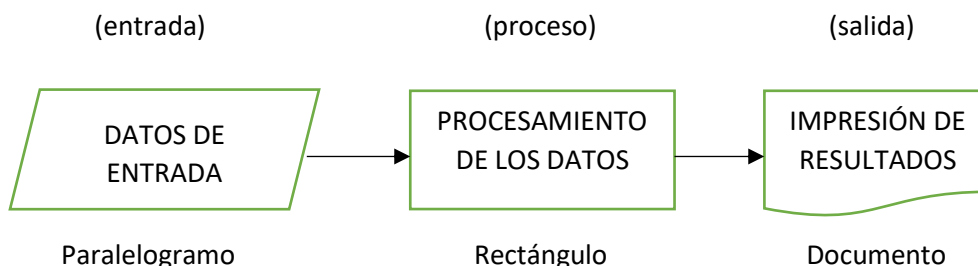
- Por otra parte, las características que deben tener los algoritmos son las siguientes:

Precisión: Los pasos a seguir en el algoritmo se deben *precisar* claramente.

Determinismo: El algoritmo, dado un conjunto de datos de entrada idénticos, siempre debe arrojar los mismos resultados.

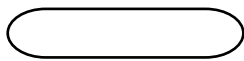
Finitud: El algoritmo, independiente de la complejidad del mismo, siempre debe tener longitud finita.

- El algoritmo consta de tres secciones o módulos principales.

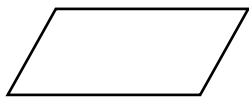


1.2 Diagramas de flujo

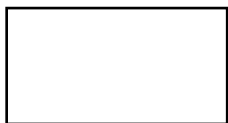
- El diagrama de flujo representa la esquematización gráfica de un algoritmo.
- En realidad, muestra gráficamente los pasos o procesos a seguir para alcanzar la solución de un problema.
- La construcción correcta del mismo es muy importante, ya que a partir de éste se escribe el programa en un lenguaje de programación determinado.
- En este caso utilizaremos el lenguaje **C**, aunque cabe recordar que el diagrama de flujo se debe construir de manera independiente al lenguaje de programación.
- El diagrama de flujo representa la solución del problema.
- El programa representa la implementación en un lenguaje de programación.
- A continuación, los símbolos utilizados en los diagramas de flujo.



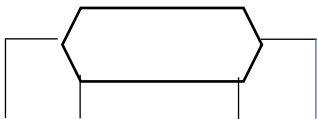
Se utiliza para marcar el *inicio* y el *fin* del diagrama de flujo.



Se utiliza tanto, para introducir los datos de *entrada*, así como para escribir los datos de *salida* o sean los resultados.



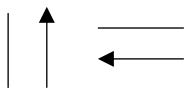
Representa un *proceso*. En su interior se colocan asignaciones, operaciones aritméticas, cambio de valor de celdas en memoria, etc.



Se utiliza para representar una decisión múltiple, switch. En su interior se almacena un selector y dependiendo del valor de dicho selector, se sigue por una de las ramas o caminos alternativos.



Se utiliza para representar la impresión de un resultado. Expresa *escritura*, se utiliza, sobre todo, si la impresión es en papel.



Expresan la dirección del flujo del diagrama.



Expresa conexión dentro de una misma página.

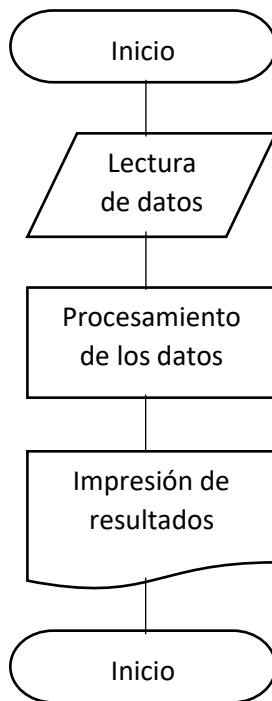


Representa conexión entre páginas diferentes.



Se utiliza para expresar un módulo de un problema, sub-Problema, que hay que resolver antes de continuar con el flujo diagrama.

- Presentamos los pasos que se deben seguir en la construcción de un diagrama de flujo.



El procesamiento de los datos generalmente está relacionado con el proceso de toma de decisiones. Además, es muy común repetir un conjunto de pasos.

- Ahora, veamos las reglas para la construcción de diagramas de flujo.
 1. Todo diagrama de flujo debe tener un **inicio** y un **fin**.
 2. Las líneas utilizadas para indicar la dirección del flujo del diagrama deben ser rectas: verticales u horizontales.
 3. Todas las líneas utilizadas para indicar la dirección del flujo del diagrama deben estar conectadas. La conexión puede ser a un símbolo que exprese lectura, proceso, decisión, impresión, conexión o fin del diagrama.
 4. El diagrama de flujo debe construirse de arriba hacia abajo (*top-down*) y de izquierda a derecha (*right to left*).
 5. La notación utilizada en el diagrama de flujo debe ser independiente del lenguaje de programación. La solución presentada se debe escribir posteriormente en diferentes lenguajes de programación.
 6. Al realizar una tarea compleja, es conveniente poner comentarios que expresen o ayuden a entender lo que hayamos hecho.
 7. Si la construcción del diagrama de flujo requiera más de una hoja, debemos utilizar los conectores adecuados y enumerar las páginas correspondientes.
 8. No puede llegar más de una línea a un símbolo determinado.

1.3 Tipos de datos

- Los datos que procesa una computadora se clasifican en **simples** y **estructurados**.
- La principal característica de los tipos de datos *simples* es que ocupan sólo una casilla de memoria.
- Dentro de este grupo de datos *simples*, se encuentran principalmente los **enteros**, los **reales** y los **caracteres**.
- A continuación, los tipos de datos *simples*.

Tipo de datos en C	Descripción	
int	Enteros	
float	Reales	
long	Enteros de largo alcance	
double	Reales de doble precisión	
char	Carácter	Símbolos del abecedario, números o símbolos especiales, que van encerrados entre comillas.

- Por otra parte, los datos *estructurados* se caracterizan por el hecho de que con un nombre se hace referencia a un grupo de casillas de memoria.
- Un dato *estructurado* tiene varios componentes. Los **arreglos**, **cadena de caracteres** y **registros** representan los datos *estructurados* más conocidos.

1.3.1 Identificadores

- Los datos que procesará una computadora ya sean simples o estructurados, se deben almacenar en casillas o celdas de memoria para utilizarlos posteriormente.
- A estas casillas o celdas de memoria se le asigna un nombre para reconocerlas: un **identificador**, el cual se forma por medio de letras, dígitos y el carácter de subrayado (_).
- Siempre hay que comenzar con una letra.
- El lenguaje de programación c distingue entre minúsculas y mayúsculas, por lo tanto, AUX y Aux son dos identificadores diferentes.
- La longitud más común de un identificador es de tres caracteres y generalmente no excede los siete caracteres.
- En C, dependiendo del compilador que se utilice, es posible generar identificadores más grandes (con más caracteres).
- Cabe destacar que hay nombres que no se pueden utilizar por ser palabras reservadas del lenguaje C.
- Ejemplos de palabras reservadas son char, int, float, do, while, if, for, entre otras.

1.3.2 Costantes

- Las constantes son datos que no cambian durante la ejecución del programa.
- Para nombrar las constantes utilizamos identificadores.
- Existen tipos de constantes de todos los tipos de datos, por lo tanto, puede haber constantes de tipo entero, real, carácter, cadena de caracteres, etc.
- Las constantes se deben definir antes de comenzar el programa principal y éstas no cambiarán su valor durante la ejecución del mismo.
- Existen dos formas básicas de definir las constantes:

```
const int nu1 = 20;          /* nu1 es una constante de tipo entero */
const int nu2 = 15;          /* nu2 es una constante de tipo entero */
const float re1 = 2.18;      /* re1 es una constante de tipo real */
const char ca1 = "UTP";      /* ca1 es una constante de tipo caracter */
```

- Otra alternativa es la siguiente:

```
#define nu1 = 20             /* nu1 es una constante de tipo entero */
#define nu2 = 15             /* nu2 es una constante de tipo entero */
#define re1 = 2.1 ;          /* nu2 es una constante de tipo real */
#define ca1 = "UTP"          /* nu2 es una constante de tipo caracter */
```

- Otra forma de nombrar constantes es utilizando el método enumerador: **enum**.
enum {va0, va1, va2, va3}; /* define cuatro constantes enteras */

Esta definición es similar a realizar lo siguiente:

```
const int va0 = 0;
const int va1 = 1;
const int va2 = 2;
const int va3 = 3;
```

1.3.3 Variables

- Las variables son objetos que pueden cambiar su valor durante la ejecución de un programa.
- Para nombrar las variables también se utilizan identificadores.
- Al igual que en el caso de las constantes, pueden existir tipos de variables de todos los tipos de datos.
- Por lo general, las variables se declaran en el programa principal y en las funciones y pueden cambiar su valor durante la ejecución del programa.
- Observemos a continuación la forma como se declaran.

```
void main(void)
{
    . . .

    int va1, va2;          /* Declaración de variable de tipo entero */
    float re1, re2;        /* Declaración de variable de tipo real */
    char ca1, ca2;         /* Declaración de variable de tipo caracter */
    . . .
}
```

- Una vez que se declaran las variables, éstas reciben un valor a través de un **bloque de asignación**.
- La asignación es una operación destructiva. Esto significa que si la variable tenía un valor, éste se destruye al asignar el nuevo valor.

- El formato de la asignación es el siguiente:

variable = expresión o valor;

Donde expresión puede representar el valor de una expresión aritmética, constante o

variable. Observe que la instrucción finaliza con punto y coma (;).

- Analicemos a continuación el siguiente caso, donde las variables reciben un valor a través de un bloque de asignación.

```
void main(void)
{
    . . .

    int va1, va2;           /* Declaración de variable de tipo entero */
    float re1, re2;        /* Declaración de variable de tipo real */
    char ca1, ca2;         /* Declaración de variable de tipo caracter */
    . . .

    va1 = 10;              /* Asignación del valor 10 a la variable entera va1 */
    va2 = va1 + 15;        /* Asignación del valor 25 (expresión aritmética) a la variable va2 */
    va1 = 15;              /* La variable va1 modifica su valor */
    re1 = 3.235;           /* Asignación del valor 3.235 a la variable real re1 */
    re2 = re1;             /* La variable re2 toma el valor de la variable re1 */
    ca1 = 't';             /* Asignación del carácter 't' a la variable ca1 */
    ca2 = '?';             /* Asignación del carácter '?' a la variable ca2 */
    . . .
}
```

- Otra forma de realizar la asignación de un valor a una variable es cuando se realiza la declaración de la misma. Observemos el siguiente caso.

```
void main(void)
{
    . . .

    int va1 = 10, va2 = 15;
    float re1 = 3.25, re2 = 6.485;
    char ca1 = 't', ca2 = 's';
    . . .
}
```

{

- Finalmente es importante destacar que los nombres de las variables deben ser representativos de la función que cumplen en el programa.

1.4 Operadores

- Los operadores son necesarios para realizar operaciones.
- Distinguimos entre operadores **aritméticos, relacionales y lógicos**.
- Analizaremos también operadores *aritméticos simplificados*, operadores de *incremento y decremento* y el operador *coma*.

1.4.1 Operadores aritméticos

- Los operadores aritméticos nos permiten realizar operaciones entre operandos: números, constantes o variables.
- El resultado de una operación aritmética siempre es un número.
- Los operadores aritméticos son los siguientes:

Suma (+) Resta (-) Multiplicación (*) División (/) Módulo(residuo) (%)

- Al evaluar expresiones que contienen operadores aritméticos, debemos respetar la jerarquía de los operadores y aplicarlos de izquierda a derecha.
- Si una expresión contiene sub-expresiones entre paréntesis, estas se evalúan primero.
- Jerarquía de los operadores aritméticos de mayor a menor en orden de importancia:

Multiplicación (*) División (/) Módulo (%) (residuo) Suma (+) Resta (-)

1.4.2 Operadores aritméticos simplificados

- Un aspecto importante del lenguaje C es la forma como se puede simplificar el uso de los operadores aritméticos.
- Considere que las variables **x** y **y** son de tipo entero (int x, y;), veamos algunos ejemplos de operadores aritméticos simplificados. Suponga que x = 7, y = 2;

Operador **Forma**

Aritmético	simplificada	Ejemplos	Equivalencia	Resultados
+	+=	x += 2;	x = x + 2;	x = 9
+	+=	x += y;	x = x + y;	x = 11
-	-=	x -= 2;	x = x - 2;	x = 5
-	-=	x -= y;	x = x - y;	x = 3
*	*=	x *= 2;	x = x * 2;	x = 14
+	*=	x *= y;	x = x * y;	x = 28
/	/=	x /= 2;	x = x / 2;	x = 3
/	/=	x /= y;	x = x / y;	x = 1

%	%=	x %= 2;	x = x % 2;	x = 1
%	%=	x %= y;	x = x % y;	x = 0

1.4.3 Operadores de incremento y decremento

- Los operadores de incremento (++) y decremento (--) son propios del lenguaje C.
- Su aplicación es muy importante porque simplifica y clarifica la escritura de los programas.
- Se pueden utilizar antes o después de las variables, siendo los resultados diferentes.
- Considere que las variables **x** y **y** son de tipo entero (int x, y;), veamos algunos ejemplos de operadores de incremento y decremento. Suponga que x = 7, y = 2;

Operador	Operación	Ejemplos	Resultados
++	Incremento	y = x++;	y = 7
			x = 8
++	Incremento	y = ++x;	y = 8
			x = 8
--	Decremento	y = x--;	y = 7
			x = 6
--	Decremento	y = --x;	y = 6
			x = 6

1.4.4 Expresiones lógicas

- Las expresiones lógicas o booleanas, llamadas así en honor del matemático George Boole, están constituidas por números, constantes o variables y operadores lógicos o relacionales.
- El valor que pueden tomar estas expresiones es 1 (en caso de ser verdadero) ó 0 (en caso de ser falsas).
- Se utilizan frecuentemente tanto en las estructuras selectivas como en las repetitivas.
- En las estructuras selectivas se emplean para seleccionar un camino determinado, dependiendo del resultado de la evaluación.
- En las estructuras repetitivas se usan para determinar básicamente si se continua con el ciclo o se interrumpe el mismo.

1.4.5 operadores relacionales

- Los operadores relacionales se utilizan para comparar dos operandos, que pueden ser números, caracteres, cadena de caracteres, constantes o variables.
- Las constantes o variables, a su vez, pueden ser de los tipos expresados anteriormente.
- A continuación, presentamos los operadores relacionales. Considere que **res** es una variable de tipo entero (int res;).

Operador	Operación	Ejemplos	Resultados
----------	-----------	----------	------------

==	Igual a	res = 'h' == 'p';	res = 0
!=	Diferente de	res = 'a' == 'b';	res = 1
<	Menor que	res = 7 < 15;	res = 1
>	Mayor que	res = 22 > 11;	res = 1
<=	Menor o igual que	res = 15 <= 2;	res = 0
>=	Mayor o igual que	res = 35 >= 20;	res = 1

- Cabe destacar que cuando se utilizan los *operadores relacionales* con *operandos lógicos*, **falso siempre es menor a verdadero**. Veamos el siguiente caso.
- res = (7 > 8) > (9 > 6); /* 0 > 1 (falso) res = 0 */

1.4.6 Operadores lógicos

- Por otra parte, los operadores lógicos los cuales permiten formular condiciones complejas a partir de condiciones simples, son de conjunción (&&), disyunción (||) y negación (!) .
- Considere que las variables **x** y **y** son de tipo entero (int x, y;), veamos algunos ejemplos de operadores lógicos.

Operador	Operación	Ejemplos	Resultados
!	Negación	x = (! (7 > 15)); /* (!0) → 1 */	x = 1
!	Negación	y = (!0)	y = 1
&&	Conjunción	x = (35 > 20) && (20 <= 23); /* 1 && 1 */	x = 1
&&	Conjunción	y = 0 && 1;	y = 0
	Disyunción	x = (35 > 20) (20 <= 18); /* 1 0 */	x = 1
	Disyunción	y = 0 1;	y = 1


1.4.7 El operador coma

- La coma (,) utilizada como operador sirve para encadenar diferentes expresiones.
- Consideremos que las variables **x**, **y**, **z** y **v** son de tipo entero (int x, y,z,v;), observemos a continuación diferentes casos en la siguiente tabla.

Expresión	Equivalencia	Resultados
x = (v = 3, v * 5);	v = 3;	v = 3
	x = v * 5;	x = 15
x = (v +=, v % 3);	v = v + 5;	v = 8
	x = v % 3;	x = 2
x = (y = (15 > 10), z = (2 >= y), y && z);	y = (15 > 10);	y = 1
	z = (2 >= y);	z = 1
	x = y && z;	x = 1
x = (y = (! (7 > 15)), z = (35 > 40) && y, (! (y && z)));	y = (! (7 > 15));	y = 1
	z = (35 > 40) && y;	z = 0
	X = (! (y && z));	x =

1.4.8 Prioridades de los operadores

- Por último y luego de haber presentado los diferentes operadores (aritméticos, relacionales y lógicos), se muestra la tabla de jerarquía de los mismos.
- Cabe destacar que en C, las expresiones se evalúan de izquierda a derecha, pero los operadores se aplican según su prioridad.

Operadores	Jerarquía
()	(mayor)
!, ++, --	
*, /, %	
+, -	
==, !=, <, >, <=, >=	
&&,	
+=, -=, *=, /=, %=	
,	
	(menor)

- El operador () es asociativo y tiene la prioridad más alta en cualquier lenguaje de programación.