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 puede 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 requiriera 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 se encuentran principalmente los **enteros**, los **reales** y los **caracteres**.

TABLA 1.2. Tipos de datos simples

Tipo de datos en C	*Descripción*	*Rango*
`int`	Enteros	-32,768 a +32,767
`float`	Reales	3.4×10^{-38} a 3.4×10^{38}
`long`	Enteros de largo alcance	-2'147,483,648 a 2'147,483,647
`double`	Reales de doble precisión	1.7×10^{-308} a 1.7×10^{308}
`char`	caracter	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. Es decir, un dato estructurado tiene varios componentes. Los **arreglos**, **cadena de caracteres** y

registros representan los datos estructurados más conocidos. Éstos se estudiarán a partir del capítulo 4.

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 les asigna un nombre para reconocerlas: un **identificador**, el cual se forma por medio de letras, dígitos y el caracter 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**. Estos nombres prohibidos se presentan en la siguiente tabla.

TABLA 1.3. Palabras reservadas del lenguaje C

`auto`	`do`	`goto`	`signed`	`unsigned`
`break`	`double`	`if`	`sizeof`	`void`
`case`	`else`	`int`	`static`	`volatile`
`char`	`enum`	`long`	`struct`	`while`
`const`	`extern`	`register`	`switch`	
`continue`	`float`	`return`	`typedef`	
`default`	`for`	`short`	`union`	

1.3.2. Constantes

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, caracter, 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 = 'f';        /* 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 2.18             /* re1 es una constante de tipo real. */
#define ca1 'f'              /* ca1 es una constante de tipo caracter. */
```

Otra forma de nombrar constantes es utilizando el método enumerador: `enum`. Los valores en este caso se asignan de manera predeterminada en incrementos unitarios, comenzando con el cero. `enum` entonces es útil cuando queremos definir constantes con valores predeterminados. A continuación se presenta la forma como se declara un `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 (como veremos en la sección 1.6 y en el capítulo 4, respectivamente), 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 variables de tipo entero. */
float re1, re2;              /* Declaración de variables de tipo real. */
char ca1, ca2;               /* Declaración de variables 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. Observa 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;
float re1, re2;
char ca1, ca2;
...
va1 = 10;          /* Asignación del valor 10 a la variable va1.*/
va2 = va1 + 15;    /* Asignación del valor 25 (expresión aritmética) a 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 caracter 't' a la variable ca1. */
ca2 = '?';         /* Asignación del caracter '?' 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. Dado que C distingue entre los tipos de operandos (`int` y `float`) que se utilizan en una operación aritmética, en la tabla 1.4 se presentan los operadores aritméticos, varios ejemplos de su uso y el resultado correspondiente para cada uno de estos casos. Es importante observarlos cuidadosamente. Considera que `x` es una variable de tipo entero (`int x`) y `v` es una variable de tipo real (`float v`).

TABLA 1.4. Operadores aritméticos

Operador aritmético	*Operación*	*Ejemplos*	*Resultados*
+	Suma	`x = 4.5 + 3;`	`x = 7`
		`v = 4.5 + 3;`	`v = 7.5`
-	Resta	`x = 4.5 - 3;`	`x = 1`
		`v = 4.5 - 3;`	`v = 1.5`
*	Multiplicación	`x = 4.5 * 3;`	`x = 13`
		`v = 4.5 * 3;`	`v = 13.5`
		`v = 4 * 3;`	`v = 12.0`
/	División	`x = 4 / 3;`	`x = 1`
		`x = 4.0 / 3.0;`	`x = 1`
		`v = 4 / 3;`	`v = 1.0`
		`v = 4.0 / 3;`	`v = 1.33`
		`v = (float) 4 / 3;`	`v = 1.33`
		`v = ((float) 5 + 3) / 6;`	`v = 1.33`
%	Módulo(residuo)	`x = 15 % 2;`	`x = 1`
		`v = (15 % 2) / 2;`	`v = 0.0`
		`v = ((float) (15 % 2)) / 2;`	`v = 0.5`

Al evaluar expresiones que contienen operadores aritméticos debemos respetar la jerarquía de los operadores y aplicarlos de izquierda a derecha. Si una expresión contiene subexpresiones entre paréntesis, éstas se evalúan primero. En la tabla 1.5 se presenta la jerarquía de los operadores aritméticos de mayor a menor en orden de importancia.

TABLA 1.5. Jerarquía de los operadores aritméticos

Operador	*Operación*
`*, /, %`	Multiplicación, división, módulo
`+, -`	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. En la tabla 1.6 se presentan los operadores aritméticos, la forma simplificada de su uso, ejemplos de aplicación y su correspondiente equivalencia. Considere que `x` y `y` son variables de tipo entero (`int x, y`).

TABLA 1.6. Operadores aritméticos: forma simplificada de uso

Operador aritmético	*Forma simplificada de uso*	*Ejemplos*	*Equivalencia*	*Resultados*
`+`	`+=`	`x = 6;`	`x = 6;`	`x = 6`
		`y = 4;`	`y = 4;`	`y = 4`
		`x += 5;`	`x = x + 5;`	`x = 11`
		`x += y;`	`x = x + y;`	`x = 15`
`-`	`-=`	`x = 10;`	`x = 10;`	`x = 10`
		`y = 5;`	`y = 5;`	`y = 5`
		`x - = 3;`	`x = x - 3;`	`x = 7`
		`x - = y;`	`x = x - y;`	`x = 2`
`*`	`*=`	`x = 5;`	`x = 5;`	`x = 5`
		`y = 3;`	`y = 3;`	`y = 3`
		`x *= 4;`	`x = x * 4;`	`x = 20`
		`x *= y;`	`x = x * y;`	`x = 60`

continúa

TABLA 1.6. Continuación

Operador aritmético	*Forma simplificada de uso*	*Ejemplos*	*Equivalencia*	*Resultados*
/	/=	`x = 25;`	`x = 25;`	`x = 25`
		`y = 3;`	`y = 3;`	`y = 3`
		`x /= 3;`	`x = x / 3;`	`x = 8`
		`x /= y;`	`x = x / y;`	`x = 2`
%	%=	`x = 20;`	`x = 20;`	`x = 20`
		`y = 3;`	`y = 3;`	`y = 3`
		`x %= 12;`	`x = x % 12;`	`x = 8`
		`x %= y;`	`x = x % y;`	`x = 2`

1.4.3. Operadores de incremento y decremento

Los operadores de **incremento** (++) y **decremento** (--) son propios del lenguaje **C** y su aplicación es muy importante porque simplifica y clarifica la escritura de los programas. Se pueden utilizar antes o después de la variable. Los resultados son diferentes, como se puede observar en los ejemplos de la tabla 1.7. Considera que `x` y `y` son variables de tipo entero (`int x, y`).

TABLA 1.7. Operadores de incremento y decremento

Operador	*Operación*	*Ejemplos*	*Resultados*
++	Incremento	`x = 7;`	`x = 7`
		`y = x++;`	`y = 7`
			`x = 8`
		`x = 7;`	`x = 7`
		`y = ++x;`	`y = 8`
			`x = 8`
--	Decremento	`x = 6;`	`x = 6`
		`y = x--;`	`y = 6`
			`x = 5`
		`x = 6;`	`x = 6`
		`y = --x;`	`y = 5`
			`x = 5`

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 verdaderas— o `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 continúa 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, cadenas de caracteres, constantes o variables. Las constantes o variables, a su vez, pueden ser de los tipos expresados anteriormente. A continuación, en la tabla 1.8, presentamos los operadores relacionales, ejemplos de su uso y el resultado de dichos ejemplos. Considera que `res` es una variable de tipo entero (`int res`).

TABLA 1.8. Operadores relacionales

Operador relacional	*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)      ⇒ 0   */
```

El valor de `res` es igual a `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 (`!`). En la tabla 1.9 se presentan los operadores lógicos, ejemplos de su uso y resultados de dichos ejemplos. Considera que `x` y `y` son variables de tipo entero (`int x, y`).

TABLA 1.9. Operadores lógicos

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

La tabla de verdad de estos operadores se presenta a continuación.

TABLA 1.10. Tabla de verdad de los operadores lógicos

P	Q	*(! P)*	*(! Q)*	*(P || Q)*	*(P && Q)*
Verdadero 1	Verdadero 1	Falso 0	Falso 0	Verdadero 1	Verdadero 1
Verdadero 1	Falso 0	Falso 0	Verdadero 1	Verdadero 1	Falso 0
Falso 0	Verdadero 1	Verdadero 1	Falso 0	Verdadero 1	Falso 0
Falso 0	Falso 0	Verdadero 1	Verdadero 1	Falso 0	Falso 0

1.4.7. El operador coma

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

TABLA 1.11. Usos del operador coma

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

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.

TABLA 1.12. Jerarquía de los diferentes operadores

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

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