Programación en Lenguaje C

Prep. Juan P. Ruiz N.

2012

Prólogo

El objetivo de esta guía es dar un enfoque práctico a lo que se enseña en el curso de Computación I de la Universidad Simón Bolívar, ilustrar un poco el proceso mental que se sigue al momento de generar un programa en solución a un problema planteado. He tratado de cubrir todos los ejemplos posibles para cada uno de los temas planteados; sin embargo, siempre existirán pequeñas dudas que solo podrán ser resueltas por medio de la práctica constante.

Esta guía no busca ser una referencia formal y mucho menos un sustituto a los textos recomendados para el curso; su intención es la de brindar un apoyo adicional que facilite la comprensión de los temas enseñados en él.

Finalmente me gustaría agradecer a la Prof. Maruja Ortega y el Prof. Roger Clotet, quienes fueran mis profesores durante los cursos de Computación I y II en la Universidad Simón Bolívar, por mostrarme lo divertido que puede ser el mundo de los algoritmos y la programación. Y al Prof. Leonid Tineo, por abrirme las puertas como preparador del Departamento de Computación y Tecnología de la Información.

Juan P. Ruiz N.
Marzo, 2012

Índice

1. Conceptos Básicos 1

2. Variables y Operadores 8

3. Instrucciones de Entrada y Salida 13

4. Estructuras Condicionales 17

5. Estructuras Iterativas 22

6. Funciones y Alcance de Variables 25

7. Tipos de Datos Estructurados 31

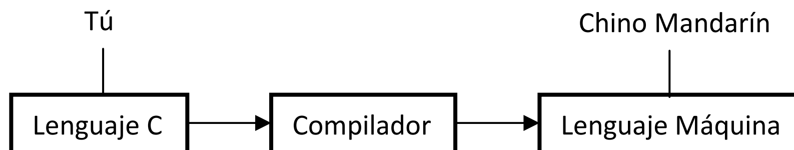
Conceptos Básicos

Cuando todos empezamos a oír sobre la programación y la creación de programas informáticos, quizá la pregunta más frecuente es “¿Cómo pasamos de palabras escritas a tener un programa ejecutable?”. El concepto de **compilación,** al igual que casi todas las herramientas presentes en la programación, puede llegar a ser bastante abstracto cuando no se ha probado personalmente, es por esto que se recomienda ampliamente el seguir esta guía al mismo tiempo que se va experimentando con cada una de las nuevas herramientas aprendidas.

Los programas en C se escriben utilizando un editor de texto común (El Notepad de Windows por ejemplo) y guardando el archivo con extensión “.c”; a este documento se le conoce como **fichero fuente**, mientras que al texto del programa que este contiene se le conoce como **código fuente**.

La pregunta más razonable en este punto sería algo así: “Pero… ¿Nuestra computadora puede entender lo que escribimos?”. La respuesta corta es NO, la computadora habla un lenguaje muy particular conocido como **lenguaje de máquina**, el cual está compuesto por instrucciones en código binario.

El **compilador** es un programa externo que se encarga de traducir el código fuente en lenguaje de máquina para que nuestra computadora pueda entender lo que le decimos que haga; y este a su vez, es empaquetado dentro de un **Archivo Ejecutable**, el cual puede finalmente ser utilizado por el usuario.



Aunque sea totalmente posible escribir un programa en lenguaje C utilizando un editor de texto cualquiera, lo más común es utilizar los llamados **IDE** (Integrated Development Environment). Un IDE es un programa compuesto por un conjunto de herramientas que facilitan el trabajo del programador; como lo son: un editor de texto especialmente diseñado para identificar las distintas partes de un código fuente, un compilador, y en casos de los más avanzados, también un depurador y un editor de interfaz grafica. Para el resto de esta guía se recomienda el uso del IDE “Code::Blocks” el cual puede ser descargado con licencia gratuita desde su página oficial.

Importante: Code::Blocks viene en dos versiones, la primera incluye el compilador GCC, la segunda no incluye compilador. Para poder compilar los programas escritos dentro del IDE es necesario descargar la versión con compilador incluido (aproximadamente 70MB).

Como compilar un archivo fuente

Una vez escrito nuestro código fuente debemos saber cómo compilarlo utilizando Code::Blocks.

1. Abrimos nuestro archivo fuente por medio de la barra de herramientas file->open, o bien utilizando el atajo Ctrl+o.
2. Una vez abierto nuestro archivo procedemos a compilarlo por medio del menú build->build, o bien utilizando el atajo Ctrl+F9
3. Una vez hemos creado nuestro ejecutable podemos proceder a ejecutarlo por medio del menú build->run, o por el atajo Ctrl+F10

Al realizar estos pasos se abrirá una ventana con la consola de Windows ejecutando nuestro nuevo programa.

Algoritmos Secuenciales

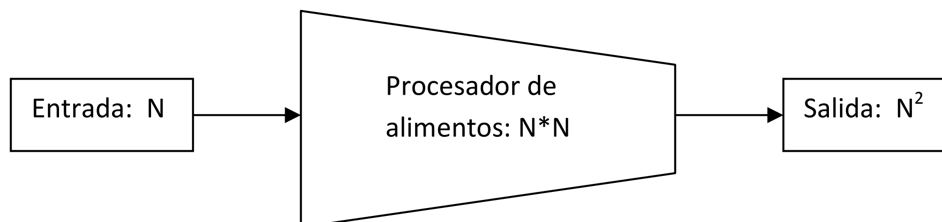
La programación, sin importar el área de aplicación, fue creada para generar soluciones a problemas de automatización de tareas, reduciendo cualquier operación a tan solo entrada y salida de datos, liberando de trabajo al ser humano. Al conjunto de pasos para solucionar dicho problema se le llama **Algoritmo.**

La definición formal de algoritmo según la RAE: "Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.". Esta definición nos da dos pistas básicas sobre el correcto diseño de un algoritmo: debe ser **ordenado** y **finito**. Las instrucciones dentro de un algoritmo deben tener un orden específico, así como un inicio y fin bien determinados.

La mejor manera de ver el diseño de algoritmos es como un procesador de alimentos, donde los valores de entrada son los alimentos y los valores de salida son los alimentos ya procesados y picados. Siempre, sin importar la naturaleza del problema, nos interesará reducirlo a un procesador de alimentos, en la cual introduzcamos todos los datos que poseemos y obtengamos como salida nuestros vegetales bien picados.

Ejemplo de algoritmo básico para calcular el cuadrado de un número:

1. Recibir número al cual se le desea aplicar el calculo
2. Multiplicar el número recibido por si mismo
3. Devolver el resultado de la operación



Pasos para diseñar un algoritmo

Los pasos básicos para el diseño de un algoritmo valido son los siguientes:

1. Definición o especificación del problema
    a. **Entradas**: Rango de validez y condiciones que deben satisfacer los datos de entrada (**Precondición**).

b. **Salidas**: Tipo y propiedades que deben cumplir los valores resultantes expresados en términos de los datos de entrada (**Postcondición**).

Ejemplo 1: Calcular el resultado de la división entre dos números

Entradas: Dos números enteros, numerador y denominador

Precondición: Denominador diferente de cero

Salida: Un numero real

Postcondición: resultado = numerador / denominador

Ejemplo 2: Determinar el máximo entre dos números dados

Entradas: Dos números X,Y reales

Precondición: Cualquier número real es valido

Salida: Un número real MAX

Postcondición: MAX mayor o igual que X, MAX mayor o igual que Y

2. Descomponer el problema en subproblemas más sencillos (**Análisis Descendente**)

Ejemplo: Calculo de admitidos en la USB

a. Calcular promedio de bachillerato
b. Obtener notas de prueba de admisión
c. Para cada estudiante calcular nota total = 0.3 * NotaBachillerato + 0.7 * NotaPruebaAdmisión
d. Obtener punto de corte
e. Ordenar notas totales de mayor a menor
f. Para cada nota mayor que el punto de corte guardar en una lista la cedula.

3. Combinar estructuras algorítmicas básicas (Secuenciales, Condicionales o Cíclicas) para resolver los subproblemas.

4. Ensamblar las soluciones a los subproblemas.

Estructura de un programa en C

Ahora que ya conocemos toda la base teórica necesaria echémosle un vistazo a nuestro primer programa:

```
/* Mi Primer Programa */

int main(){

      int numero;

      numero = 2+2;

      return 0;

}
```

No te preocupes si aun no entiendes lo que sucede dentro de este pequeño programa, todo se irá aclarando poco a poco. Iremos explicando el programa línea a línea.

1. `/* Mi Primer Programa */`

En la programación existen ilimitadas maneras de ejecutar una tarea, unas fáciles de comprender y otras no tan fáciles, es por esto que se crearon las líneas de comentarios. Cuando se introduce un comentario dentro de un programa, el compilador lo reconoce como tal y lo ignora al momento de hacer la traducción.

Hay dos maneras de indicarle al compilador que lo que escribimos es un comentario. La primera es la que vemos en el ejemplo de arriba, encerrando el texto **/*** en barras y asteriscos ***/**, permitiendo incluso hacer saltos de líneas sin necesidad de terminar el comentario.

```
/* Esto también

Es

Un

Comentario valido */
```

La segunda manera de realizar un comentario es con el uso de doble barras **//** al inicio del texto que deseamos que el compilador ignore. La diferencia entre método y el anterior es que en este caso el compilador solo interpretará como comentario lo que siga después de la doble barra y por el resto de la línea, sin la posibilidad de realizar saltos.

```
// Esto es un Comentario

Esto NO es un comentario

Esto NO es comentario //Pero esto sí lo es
```

2. `int main(){`

Esta línea es muy especial, ya que la encontraras en todos los programas de C, así como en muchos otros lenguajes de programación. Se trata de la declaración de una función, en este caso la función principal del programa. La palabra reservada **int** se refiere al valor de retorno de la función como un número de la familia de los enteros (**int**eger), mientras que la palabra **main** se refiere al nombre propio de la función.

En un principio todos los programas que escribas en C deben poseer esta línea, ya que es la que le indica al compilador donde iniciar el programa, y en su debido momento también le indicará donde terminar.

3. `int numero;`

Este es nuestro primer encuentro con una instrucción, en este caso en particular con una instrucción de declaración de variable. Con esta línea le estamos indicando al compilador que usaremos una variable llamada "numero" en la cual almacenaremos valores del tipo entero. Adicionalmente, toda instrucción debe finalizar con punto y coma.

4. `numero = 2+2;`

Otra instrucción, esta vez estamos diciéndole al compilador que sume 2+2 y guarde el resultado dentro de "numero". No haré demasiado énfasis en esto por ahora, profundizaremos en las declaraciones y asignaciones en el próximo capítulo.

5. `return 0;`

Nuevamente nos encontramos con una instrucción esta vez utilizamos la palabra reservada **return** la cual indica al programa que debe abandonar la ejecución de la función y continuar a partir del punto en el que se realizó la llamada. El 0 es el valor de retorno de la función. Por convención, cuando "main" devuelve un valor 0 indicamos que todo está correcto en el programa. Un valor distinto normalmente indica un error en la ejecución.

6. }

Con esta llave le estamos indicando al programa que finalizamos la función “main”. Las llaves suelen indicar el inicio y final de una función o una instrucción, aunque también pueden tener otros usos que más adelante veremos.

El único problema con este programa es, si obedecieron a la introducción de esta guía y corrieron a probarlo, habrán notado que no hace absolutamente nada. Está bien, suma correctamente dos más dos y guarda el resultado, pero nosotros no podemos verlo. Para eso hace falta trabajar con entradas y salidas de datos, las cuales veremos en un par de capítulos.

Antes de finalizar hagamos un resumen de la estructura de un programa en C:

```
/* Mi Primer Programa */ //Comentarios e introducción al programa

int main(){ //Inicio del bloque principal del programa, función main

                  //
      int numero; //Declaración de variables
                  //

                    //
      numero = 2+2; //Secuencia de instrucciones
                    //

      return 0; //Retorno de la función
}
```

Variables y Operadores

Variables

Desde el punto de vista de un programador y sin adentrarnos en lo filosófico del asunto, una variable es un elemento cuyo valor puede cambiar a lo largo de la ejecución del programa. Desde el punto de vista de nuestra computadora, una variable es un espacio dentro de la memoria que nos permite almacenar información relativa al programa que estamos desarrollando. El origen de dicha memoria no es de nuestra incumbencia.

En el capítulo anterior vimos un poco de la estructura de declaración de una variable, en este capítulo extenderemos un poco la teoría de variables y daremos algunos ejemplos.

```
//Sintaxis para declaración de variables
[Tipo] <identificador> [[,<identificador2>,<identificador3>,…]];
```

Para el resto de la guía las definiciones de sintaxis se realizaran con la siguiente convención. La información entre corchetes [] indicarán una palabra reservada que cumplirá la función que se indica dentro de los corchetes. La información entre <> corresponden a un texto que tomara la función de lo que se indica en su interior. Los doble corchetes [[ ]] indicarán una parte opcional en la sintaxis.

El **identificador** será el nombre por el que nos referiremos a la variable, será una secuencia de caracteres (letras, números y "_") que distingue entre mayúsculas y minúsculas y debe comenzar por una letra o el carácter "_".

El **tipo** se refiere a la clase de valores que será capaz de tomar la variable. Los tipos existentes son:

Números Enteros -----> **short** (poca precisión), **int** (mayor precisión)

Números Reales -------> **float** (poca precisión), **double** (mayor precisión)

Caracteres --------------> **char**

Adicionalmente, la sintaxis nos permite declarar más de una variable del mismo tipo en una sola instrucción; basta con separar los identificadores de cada variable con una coma y por supuesto nunca olvidar el ";" al finalizar la instrucción.

Ejemplos:

Declaraciones de variables validas

```
int variable;
float MiIndice, _Pi = 3,14;
char Dia,Mes,Ano;
short edad1,edad2,edad3;
```

Declaraciones de variables NO validas

```
int int; //int,float,char,etc. son identificadores reservados del lenguaje
int 1edad,2edad,3edad;
float *índice;
float Mi Indice;
char letra?
```

Operadores

Los operadores son elementos que generan ciertos cálculos al utilizarlos en conjunto con las variables. Como es de esperarse, entre ellos encontraremos los mismos utilizados en las matemáticas; sin embargo, contaremos con unos cuantos adicionales. Existen cuatro tipos de operadores: Aritméticos, de Asignación, Lógicos y Relacionales.

Aritméticos:

- + Suma
- - Resta
- * Multiplicación
- / División
- % Resto de la división

Asignación:

- = Asignación
- - = Resta y asignación
- * =Multiplicación y asignación
- / =División y asignación
- %= Resto y asignación

Lógicos:

- ! No
- && And
- || Or

Relacionales:

- < Menor que
- > Mayor que
- <= Menor o igual que
- >= Mayor o igual que
- == Igual que
- != Distinto de

Adicionalmente, existe una prioridad por parte del compilador al realizar las operaciones que es muy importante tener en cuenta, para así poder evitar errores de cálculos al momento de escribir un programa.

Prioridad de operadores:

- ()
- !
- * , / , %
- + , -
- < , <= , > , >=
- == , !=
- &&
- ||

Importante: En caso de que existan en una instrucción más de un operador con la misma prioridad, entre ellos, la prioridad será mayor para el que se encuentre más a la izquierda.

Uso de los Operadores

Al ver los operadores, es normal sentirse tentado a utilizarlos de la misma manera que estamos acostumbrados dentro del mundo de las matemáticas. Sin embargo, existen situaciones especiales para cada uno de los operadores, que hace falta aclarar antes de lanzarnos de lleno a utilizarlos en la vida real. Vayamos uno por uno.

```
//Sintaxis de operadores aritméticos

<expresión> + <expresión>

<expresión> - <expresión>

<expresión> * <expresión>

<expresión> / <expresión>

<expresión> % <expresión>
```

Tanto la **Suma**, **Resta** y **Multiplicación** funcionan exactamente de la misma manera que en matemáticas; sin embargo, la **División** tiene una pequeña variación. Existen dos métodos para realizar la división entre números, la versión para enteros y la versión para reales.

Cuando realizamos la división utilizando variables enteras, el resultado será solo la parte entera del resultado final, teniendo la posibilidad de obtener el **residuo** utilizando el operador %. Cuando en la división a menos uno de las variables utilizadas es real, obtendremos como resultado un número real, el cual incluirá una cantidad considerable de los decimales del resultado final.

División:

- 5/2 ---------> 2
- 5%2 ---------> 1
- 5.0/2 -------> 2.5
- 5/2.0 -------> 2.5
- 5.0/2.0 ----->2.5

Dejando los operadores aritméticos claros hablemos ahora del operador **Asignación**. Vimos que en el caso de los operadores aritméticos estos operaban de izquierda a derecha, pero en el caso del operador = tendremos lo contrario; él operará de derecha a izquierda, tomando todo lo que se encuentre a su derecha y almacenándolo a la variable a su izquierda.

```
//Sintaxis de operadores de asignación

<variable> <operador de asignación> <expresión>
```

Es normal confundir al operador asignación con su versión matemática, es por esto que debemos tener claro que no estamos hablando de una igualdad, sino de una asignación; por lo tanto, cuando decimos que “x = 1+2” no estamos diciendo que “x” es igual a 3, sino que “x”

AHORA será igual a 3, sin importar su valor anterior. Esto hace posible decir cosas como “x = x+1”; lo que en matemáticas te llevaría a ser condenado al exilio, aquí solo significa que al valor de “x” le sumamos 1 y se lo asignamos a “x” nuevamente. Consideremos el ejemplo siguiente:

```
//Operadores de asignación
int a=10, b=0;

b = a;
b = b + 1;
```

Vemos que la primera instrucción declara dos variables a y b a las cuales les asigna los valores 10 y 0 respectivamente. Luego le asignamos a “b” el contenido de “a” y finalmente le asignamos a “b” el contenido de “b + 1”, lo cual dejaría como valores finales de “a” y “b”, 10 y 11 respectivamente. No dejes que la frustración te consuma, si no entiendes estos resultados inmediatamente mira atentamente a la hoja por dos minutos.

Una vez entendido el operador asignación podemos hablar del resto de los operadores de asignación. Para todos ellos la formula es la misma:

```
//Sintaxis de operadores de asignación

<variable> op= <expresión>

//Expresión equivalente

<variable> = <variable> op <expresión>
```

Algunos ejemplos:

```
a += b  => a = a + b

a *= b  => a = a * b

a -= b  => a = a – b

a /= b  => a = a / b
```

Los operadores **Lógicos** y **Relacionales** los trabajaremos más adelante.

Instrucciones de Entrada y Salida

A pesar de todo lo que se ha aprendido hasta ahora, aun no hemos sido capaces de observar personalmente el comportamiento de nuestros programas, sabemos que funcionan, sabemos cómo funcionan, pero no podemos verlo. Para esto hace falta usar las funciones de entrada y salida de datos que nos proporciona muy gentilmente C. Pero para empezar a hablar de ellas primero hace falta hablar un poco sobre las instrucciones al preprocesador.

El **Preprocesador** es un programa externo que es invocado por el compilador, el cual se encarga de eliminar comentarios, incluir archivos fuente externos, entre otras funciones, todo esto antes de que empiece la traducción real. Las **Instrucciones al Preprocesador** son aquellas que nos permiten indicarle al preprocesador lo que debe hacer antes de empezar el proceso de compilación.

Las instrucciones más utilizadas son:

#include Nos permite incluir un archivo fuente externo nuestro programa, esto nos permite agregar a nuestro código funciones creadas previamente tanto por nosotros como por terceros, evitándonos así mucho trabajando.

```
#include <stdio.h>  //Ejemplo de un include
```

#define Esta instrucción nos permite definir constantes numéricas para su uso dentro del programa.

```
#define NUMERO 5  //Ejemplo de un define

variable = NUMERO +1;  //Ejemplo de eso de una constante declarada por define
```

Ahora que conocemos las instrucciones al preprocesador podemos hablar de la librería estándar **stdio.h** (standard input-output header). Es un archivo fuente utilizado en lenguaje C para la realización de operaciones de entrada y salida.

1. **Instrucción Imprimir en Pantalla**

```
//Sintaxis para el uso de printf
#include <stdio.h> //Necesario para su uso, solo es requerida su inclusión
                   //una vez por código fuente

printf("<secuencia de control>", <variables o expresiones>);
```

Veamos detalladamente la sintaxis. Primero nos encontramos con la inclusión de la librería de entrada y salida de C, esta línea solo hace falta una vez al inicio de nuestro código fuente para poder hacer uso de la instrucción de impresión tantas veces como queramos.

Luego vemos que la instrucción de escritura comienza con "printf(" y termina con ");", esta es básicamente la estructura de cualquier función en el lenguaje C, pero ya llegaremos a ese punto. Vemos que dentro de los paréntesis de nuestra función "printf();" tenemos una secuencia de control escrita entre comillas, con esto le indicamos al compilador cual es la estructura de la frase que queremos imprimir en pantalla. Dentro de la secuencia de control podemos colocar tanto texto como ciertos modificadores que se encargarán de darle forma a la salida:

Modificadores

\n -> Siguiente línea

%d ->Número entero en formato decimal

%f ->Numero Real

%c ->Caracter

Cada modificador tiene su uso específico. Empecemos con el "\n", este modificador le indica al compilador que debe insertar un salto de línea dentro de la frase que queremos imprimir. Los modificadores "%d","%f" y "%c" le indican al compilador que, en el lugar donde ellos están, se debe colocar un dato del tipo que ellos indiquen. La pregunta es ¿Cómo sabe el compilador de dónde sacará estos datos?, sencillo, se los damos nosotros por medio de la lista de variables que se introduce justo después de la secuencia de control, y el compilador se encargará de ir sustituyéndolas en los modificadores, exactamente en el mismo orden de aparición.

A continuación se muestran algunos ejemplos con las distintas posibilidades de uso de la función "printf".

Código Fuente	Consola
`printf("Hola Mundo!");`	`Hola Mundo!`

```
Código Fuente

printf(“Hola\nMundo!”);
```

```
Consola

Hola
Mundo!
```

```
Código Fuente

int a = 20;
printf(“El valor de a es %d”,a);
```

```
Consola

El valor de a es 20
```

```
Código Fuente

printf(“Numeros del 1 al 5\n”);
printf(“%d,%d,%d,%d,%d”,1,2,3,4,5);
```

```
Consola

Numeros del 1 al 5
1,2,3,4,5
```

```
Código Fuente

printf(“2+2 es igual a %d”,2+2);
```

```
Consola

2+2 es igual a 4
```

2. **Instrucción Lectura de Pantalla**

```
//Sintaxis para el uso de printf
#include <stdio.h> //Necesario para su uso, solo es requerida su inclusión
                   //una vez por código fuente

scanf(“<secuencia de control>”, &<variables>);
```

Como se puede apreciar la sintaxis para lectura es prácticamente igual que la de escritura, la única diferencia que existe es que esta vez se debe agregar un ampersand “&” antes de cada variable donde deseemos almacenar los valores leídos. El resto de la sintaxis se mantiene, incluyendo los modificadores.

A continuación se muestran algunos ejemplos con las distintas posibilidades de uso de la función “scanf”.

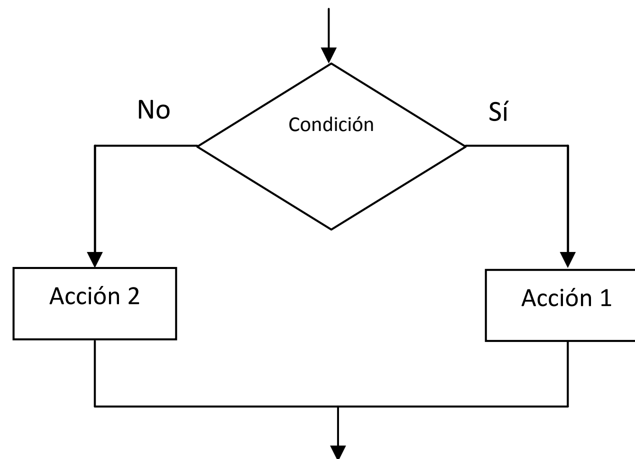
```
Código Fuente

#include <stdio.h>

int main(){

int a; //Variable entera
float b; //Variable real
char c; //Variable carácter

scanf("%d",&a); //Leer un entero

scanf("%f",&b); //Leer un real

scanf("%c",&c); //Leer un carácter

scanf("%a %b %c",&a,&b,&c); //Leer sucesivamente un entero, un real y un carácter

}
```

Con esto finaliza la primera parte de la guía, los invito a probar por ustedes mismos cada una de las cosas que se les ha enseñado hasta el momento, asegurándose de entender muy bien el funcionamiento de cada una, usando el printf y scanf como apoyo para interactuar con su programa y observar detalladamente lo que ocurre.

Puede pasar que su programa se ejecute perfectamente pero no se comporte de la manera que ustedes esperan, arrojando datos extraños o errados. La mejor manera de solucionar este tipo de problemas es utilizando la función printf para imprimir las variables en los distintos estados del programa y observar que se comporten tal cual como ustedes desean. A esto se le conoce como **Depuración,** y es una técnica extremadamente útil en el mundo de la programación.

Estructuras Condicionales

Existen situaciones durante el desarrollo de algoritmos en los que necesitamos tomar decisiones. ¿Es el valor leído negativo o positivo? ¿La torta está cruda o cocida? ¿Llegué a la universidad o debo seguir conduciendo?. En la programación este tipo de decisiones se toman por medio de estructuras condicionales.




```
//Sintaxis de una estructura condicional
if (<condición>)
  {Acción 1}
else
  {Acción 2}
```

El uso de esta instrucción es de la forma siguiente. Se utilizan las palabras reservadas por el lenguaje C, **if** y **else**, para determinar el bloque de instrucciones a seguir en el caso de cumplirse o no, respectivamente, la condición bajo la cláusula **if**. La condición por motivos obvios siempre debe estar presente; sin embargo, la cláusula **else** puede o no ser incluida.

Si durante la corrida del programa el compilador no encuentra una clausula else, en caso de no cumplirse la condición dentro del if, el programa continuará su camino como si el condicional no existiese. En el caso de encontrarse la cláusula else, entonces el programa procederá a ejecutar las instrucciones bajo esta si la condición no se cumple.

Puede parecer un poco enredado a primera vista, pero una vez vistos un par de ejemplos todo quedará más claro.

Operadores Lógicos y Relacionales

Todas las condiciones que necesiten ser escritas en el lenguaje C deben ser escritas mediante el uso de operadores lógicos y relacionales, los cuales fueron presentados durante la sección de operadores. En esta oportunidad se explicará el funcionamiento de los mismos.

Cuando se escribe una expresión de condición se espera tener como resultado si la misma es **Verdadera** o por el contrario, es **Falsa**. Estos dos posibles resultados son representados por medio de los llamados "Valores Booleanos", que en el mundo de la programación conocemos como **TRUE** y **FALSE**.

Cuando utilizamos variables relacionales para establecer una relación entre variables y/o constantes lo que obtenemos como resultado a estas expresiones es un valor booleano. Por ejemplo, si tenemos la expresión "2<3" obtendremos como resultado un "TRUE"; sin embargo, si tenemos la expresión "3<2" nuestro resultado será "FALSE". Lo mismo ocurre para el resto de los operadores relacionales.

La siguiente parada en el tren de pensamiento necesariamente es ¿Pero, como lucen estos valores en el lenguaje C? La respuesta no es sencilla, pero trataré de simplificarla lo más posible. Al contrario que en otros lenguajes, donde simplemente podemos asignar **true** o **false** a una variable declarada como booleana, C no posee una manera exacta de representar los valores booleanos. C reconocerá como un valor TRUE cualquier número entero positivo, mientras el cero será reconocido como FALSE. Considerando esto, sabemos que el condicional "if(100)" siempre se cumplirá.

Ahora solo nos queda un último punto por aclarar, los operadores lógicos. ¿Qué sucede si necesito que se cumplan no una, sino dos condiciones simultáneamente para poder realizar una acción? Podrían simplemente colocar un if dentro de otro y así se estarían verificando ambas condiciones, pero existe una forma mucho más fácil.

Los operadores lógicos no trabajan con números o letras propiamente, sino con valores booleanos, y operan de la siguiente manera:

Tabla de la Verdad					
A	B	A&&B	A||B	!A	!B
FALSE	**FALSE**	**FALSE**	**FALSE**	**TRUE**	**TRUE**
FALSE	**TRUE**	**FALSE**	**TRUE**	**TRUE**	**FALSE**
TRUE	**FALSE**	**FALSE**	**TRUE**	**FALSE**	**TRUE**
TRUE	**TRUE**	**TRUE**	**TRUE**	**FALSE**	**FALSE**

La explicación detrás de esta tabla es bastante sencilla y se puede deducir por los nombres de los operadores. Esto queda de su parte.

Veamos algunos ejemplos de condiciones:

```
2 < 3      ---> TRUE       (2 < 3) && (2 < 4)     ---> TRUE
2 < (3-1)  ---> FALSE      (2 < 3) && (3 < 2)     ---> FALSE
2 <= (3-1) ---> TRUE       (2 < 3) || (3 < 2)     ---> TRUE
2 == 2     ---> TRUE       (3 < 2) || (4 < 2)     ---> FALSE
2 == 3     ---> FALSE           !(2 < 3)          ---> FALSE
2 != 3     ---> TRUE            !(3 < 2)          ---> TRUE
2 != 2     ---> FALSE      !((2 < 3) || !(2 < 4)) ---> FALSE
```

Instrucción al Preprocesador: define

Una buena manera de designar un valor determinado para las los valores booleanos, es por medio de la definición de constantes en nuestro programa. Las constantes en lenguaje C utilizando la directiva al preprocesador "**define"**.

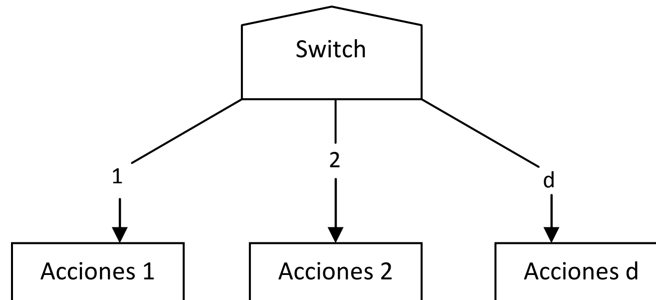
```
#define TRUE 1
#define FALSE 0
#define N 10
#define A 5
#define B 20
```

Una vez definida una constante, esta es válida por el resto del programa, y se utiliza como cualquier otro valor dentro de él.

```
int n;

n = TRUE;
n = N;
n = A + B;
```

Instrucciones: switch, case, default, break

Existen casos en los que las posibilidades de un condicional son más que verdaderas o falsas. Supongamos que somos Sheldon Cooper y necesitamos saber qué actividad nos toca en función del día actual. Para resolver este problema la mejor opción es almacenar en una variable el día actual de la semana y utilizar un switch para determinar qué actividad corresponde.



El **switch** acepta una variable como entrada y define un **case,** con su respectivo bloque de acciones, para cada posible valor que podría tomar dicha variable. El caso **default** nos permitirá determinar que debe hacer el switch para los casos no determinados por la lista de **case**.

```
//Sintaxis Switch

switch(<Variable>)
{
	case <Valor1>: <Acciones1>
		       break;

	case <Valor2>: <Acciones2>
		       break;

	default: <Acciones d>
}
```

Hay una consideración adicional cuando trabajamos con la instrucción switch. Una vez que el switch encuentra el caso correspondiente al valor actual de la variable, procederá no solo a ejecutar las acciones de dicho caso, sino las acciones de todos los casos posteriores hasta encontrarse con la instrucción **break**. El break se encargará de cortar el ciclo y sacar al programa de la instrucción switch.

De la misma manera, para asignar a varios **case** el mismo bloque de acciones, basta con colocar un caso tras otro, no colocando el bloque de acciones sino hasta llegar al último caso:

```
switch(<Variable>)
{
	case <Valor1>:
	case <Valor2>: <Acciones2>
	               break;

	default: <Acciones d>
}
```

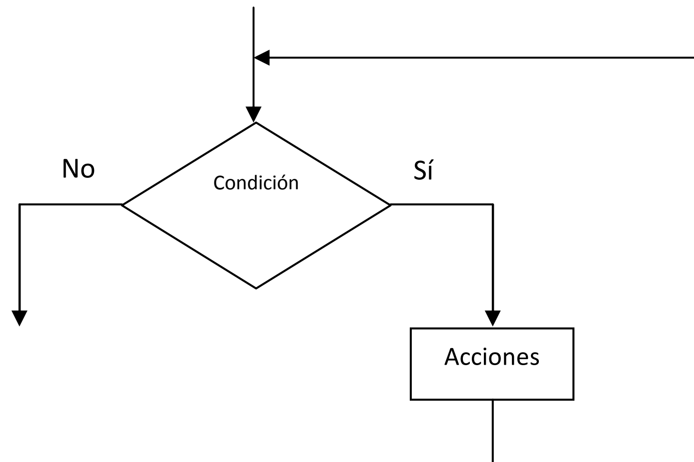
En este ejemplo el switch ejecutará las mismas acciones tanto si el valor es <Valor1> como <Valor2>. A continuación algunas posibilidades del switch.

```
int entero;
char caracter;

switch(entero)
{
	case 1: entero += 1;
	        printf("%d", entero);
	        break;

	case 2: printf("%d", entero);
	        break;

	default: printf("%d",2);
}

switch(caracter)
{
	case 'a': printf("Hola");
	          break;

	case 'b': printf("Chao");
	          break;

	default: printf("Ingrese otro caracter");
}
```

Es importante tener en cuenta que todo switch puede transformarse en un bloque de condicionales anidados, solo hace falta un poco de orden y astucia.

Estructuras Iterativas

Los ciclos son casos especiales de los condicionales. Esta vez las acciones se realizarán tantas veces como sea necesario hasta que la condición deje de cumplirse. Las condiciones tendrán la misma estructura que en el caso de los condicionales.



Una de las cosas más importantes a tener en cuenta cuando utilizamos un ciclo es la siguiente. Siempre debemos incluir dentro de las acciones alguna instrucción que rompa con la condición y nos permita salir del bucle. Recuerden que una de las condiciones para la realización de un algoritmo es la finitud.

El lenguaje C nos brinda tres instrucciones distintas para realizar ciclos, **while**, **for**, **do…while**. Cada una tiene una situación de uso distinta, pero en el fondo todas funcionan de la misma manera.

```
//Sintaxis de la instrucción while

while(<condición>)
{Acciones}
```

```
//Sintaxis de la instrucción for

for(<inicialización> ; <condición> ; <actualización>)
{Acciones}
```

```
//Sintaxis de la instrucción do… while

do
{Acciones}
while(<condición>);
```

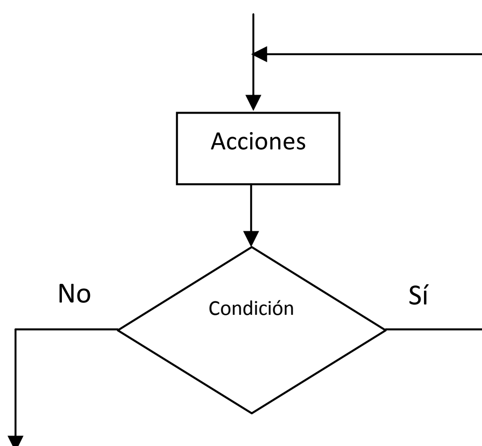
Tanto el "for" como el "do…while" tienen ambos consideraciones acerca de su uso. Fijémonos primero en la sintaxis del **for**. Vemos que tanto el inicio, el final y la actualización a las condiciones de iteración están incluidos dentro de la sintaxis; esto hace del for una estructura perfecta para su uso como contador, enumerador, o cualquier otra situación en la que conozcamos desde el inicio cuantas iteraciones se realizaran antes de finalizar el bucle. Esto no implica que no pueda ser utilizado para otras cosas, de hecho, cualquier bucle construido utilizando la instrucción while puede ser llevado a un bucle for fácilmente.

```
//Ejemplo1: Imprimir los números del 1 al 10 utilizando la instrucción while
#incluide <stdio.h>

int main(){

   int contador=1;

   while(contador <= 10){
      printf(" %d ", contador);
      contador += 1;
   }
}
```

```
//Ejemplo2: Imprimir los números del 1 al 10 utilizando la instrucción for
#incluide <stdio.h>

int main(){

   int i;

   for(i=1 ; i<=10 ; i+=1){ //Inicio del contador:1; Fin:10; Aumentar de 1 en 1
      printf("%d", i);
   }
}
```

Veamos ahora la sintaxis del **do…while**. La diferencia entre esta y las otras dos estructuras es evidente. El "do…while" nos permitirá realizar las acciones listadas bajo la instrucción **do** al menos una vez antes de verificar la condición **while**. La Utilidad de esto puede no parecer importante, pero lo es; el caso más inmediato de uso necesario de esta instrucción es en la realización de un "menú". Cuando tenemos interacciones con el usuario que incluyen su elección entre varias opciones, siempre querremos imprimir las opciones antes de verificar su decisión.

A continuación se presenta el diagrama de flujo correspondiente a la instrucción "do… while", el cual presenta una variación con respecto al presentado al inicio del capítulo.




```
//Ejemplo3: Imprimir los números del 1 al 10 utilizando la instrucción do… while
#include <stdio.h>

int main(){

   int contador=1;

   do{
      printf(" %d ", contador);
      contador += 1;
   }
   while(contador < 10) //Noten que la condición cambia, ya que al llegar a 10
                        //ya la impresión se ha realizado
}
```

Funciones y Alcance de Variables

Cuando hablábamos de los pasos para crear un algoritmo, uno de los pasos más importantes era el análisis descendente. Dijimos que el análisis descendente era la subdivisión del problema principal en pequeños problemas más sencillos, los cuales pudiéramos resolver utilizando estructuras algorítmicas simples. Por lo general nos encontraremos que estas tareas simples será necesario realizarlas más de una vez durante el desarrollo de nuestro programa. Sin embargo, existe un problema enorme con esto.

Los programadores nos caracterizamos por ser las personas más flojas del universo, lo cual tiene mucho sentido si pensamos en que nuestra tarea es crear programas que hagan las cosas por nosotros. Así que lo más sensato sería que los creadores de nuestro querido C nos dieran una herramienta para solucionar el problema de tener que escribir una y otra vez las mismas líneas de código. Por suerte, ellos piensan en todo, y nos tienen una solución maravillosa a la cual llamarón "funciones".

Las **funciones** son un conjunto de instrucciones, diseñadas para realizar una tarea específica. Adicionalmente, la sintaxis de las funciones nos permitirá especificar un conjunto de parámetros de entrada y un parámetro de salida. Espero que no hayan olvidado el ejemplo del procesador de alimentos.

```
//Sintaxis de declaración de una función

[Tipo de Salida] <Nombre> ( [Parámetros] )
{ <Instrucciones> }
```

El tipo de salida podrá ser cualquiera de los tipos de datos simples que ya conocemos. El nombre de la función cumplirá con las reglas que ya especificamos para los identificadores. Finalmente, la especificación de los parámetros cumplirá con las mismas reglas de la declaración de variables.

```
/*Ejemplo de declaración de una función que acepte 2 parámetros, una
entero y otro real. Adicionalmente tendrá una salida de tipo entero
y se llamará HolaMundo*/

int HolaMundo ( int a, float b )
{
return 0;
}
```

La instrucción **return** es utilizada dentro de las funciones para indicar cuál será el valor de salida de la función. Las funciones pueden ser declaradas tanto con un tipo de salida como con parámetros del tipo **void**.

Las funciones tienen un lugar especial dentro de nuestro código fuente, que más que una obligación es una formalidad creada por motivos de orden. Esta convención exige que todas las declaraciones de funciones se encuentren al final del código fuente, justo después de la función principal **main**. Sin embargo, existe un problema con esto, y es que C es un lenguaje secuencial, lo cual implica que para poder utilizar una función en un punto especifico del programa, es necesario que C conozca la existencia de dicha función en ese punto. Entonces, si todas las declaraciones de funciones se encuentran luego de la función main, ¿cómo podremos hacer uso de ellas?.

Como ya les dije antes, nuestros amigos de C pensaron en todo, y para este caso crearon las llamadas "Definiciones de funciones". La **definición de una función** no es más que un prototipo, un modelo de declaración de función que se coloca al inicio del programa, justo después de las instrucciones de preprocesador.

```
//Sintaxis de prototipo de función

[Tipo de salida] <Nombre> ( [Parametros] );
```

Como ven, para definir una función no hace falta más que la línea principal de la declaración, sin necesidad de colocar las acciones. Un detalle importante es que la definición lleva un punto y coma al final, mientras que la declaración no.

Finalmente, veamos como se ve todo esto dentro de nuestro programa.

```
//Programa que imprime Hola Mundo 20 veces
#include <stdio.h>

void HolaMundo (int a); //Definición de la función HolaMundo

int main ()
{
   int numero = 20;
   HolaMundo(numero); //Uso de la función HolaMundo
}

void HolaMundo (int a) //Declaración de la función HolaMundo
{
   int i;
   for(i=0;i<a;i+=1) printf("Hola Mundo\n");
}
```

Alcance de variables

Con la introducción de las funciones se nos abre el camino a toda una nueva ciencia conocida como el alcance de variables. En el lenguaje C toda variable que se declara se encuentra limitada al ambiente donde fue declarada. De momento existen dos posibles ambientes de declaración de una variable.

Las variables declaradas en el ambiente global son conocidas como **variables globales**, y podrán ser accesibles y modificables desde cualquier punto de nuestro programa. Si declaramos una variable dentro de una función, ya sea la función main o cualquier otra que hayamos creado, la variable será conocida como **variable local** y será solo accesible dentro de la función en la que se ha declarado.

Dos funciones diferentes pueden tener variables con el mismo nombre, sin embargo cada una será independiente de la otra. Adicionalmente, las variables locales solo existirán durante la ejecución de la función en la que fueron declaradas, mientras que las variables globales durarán durante toda la ejecución del programa.

Si una variable local puede ser declarada utilizando el mismo nombre que una variable local; en ese caso, todos los cambios que ocurran sobre esa variable dentro de la función serán manejados por la variable local, permaneciendo la global intacta.

Pasaje de parámetros

Cuando insertamos valores dentro de los parámetros de entrada en una función, estamos realizando un **pasaje de parámetros**. Existen dos tipos de pasaje de parámetros: Por valor y por referencia.

- Por Valor: La variable de la función toma el valor de la variable que le asignemos
- Por Referencia: La variable de la función se convierte en una instancia de la original.

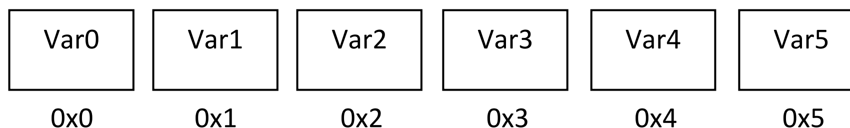
Hasta ahora hemos realizado solo pasaje de parámetros por valor. Sin embargo, existen tareas para las cuales este tipo de pasaje nos resulta muy limitado. Los invito como ejercicio a realizar una función que nos permita intercambiar los valores contenidos en dos variables distintas. Cuando hayan intentado lo suficiente pueden continuar leyendo.

Para poder hablar del pasaje de parámetros por referencia hace falta primero realizar una introducción a los **Apuntadores**.

Apuntadores

Por la forma en que está compuesta la arquitectura de procesamiento de nuestra computadora, la mejor manera de representar la memoria es por medio de "cajitas". La capacidad de almacenamiento de estas cajitas dependerá de la arquitectura del computador que estemos utilizando, pero para simplificar el modelo supondremos que cada cajita es capaz de almacenar una variable; sin embargo, siempre es importante tener presente que esto no es siempre cierto.

Un **puntero** es un tipo especial de variable capaz de contener la "dirección de memoria" de nuestra cajita. Por comodidad, trataremos a esta dirección de memoria como el "nombre" de nuestra cajita. Cada puntero será capaz de almacenar un tipo específico de cajita; ya sea **int**, **float**, **char**, o cualquier otro tipo de variable. Cual tipo de cajita apuntará nuestro puntero dependerá enteramente de nosotros y lo definiremos durante la declaración.



Bajo esta convención tenemos cinco variables, y cada una de ellas tiene su propio nombre. De esta manera, Var0 se encuentra en una cajita de nombre "0x0" y la Var4 se encuentra en una cajita de nombre "0x4".

```
//Sintaxis de declaración de un puntero

[Tipo] *<Nombre>;

//Ejemplos
int *a;
float *b,*c,d; //En esta declaración, solo b y c son punteros, d no debido a
               //que no contiene un * antes de su nombre.
char *punt;
```

Existen además dos operadores exclusivos para trabajar con apuntadores.

```
//Operadores de punteros

& -> Operador “Dirección de memoria”
* -> Operador “Contenido de lo apuntado por…”
```

El operador & nos permitirá obtener la dirección de memoria de una variable y asignársela a un puntero. Mientras que el operador * nos permitirá acceder al contenido de la variable apuntada por el puntero; este operador es totalmente independiente del asterisco utilizado para declarar al puntero.

Si mezclamos estos operadores con todos los que ya conocemos, tendremos una infinidad de instrucciones, que nos permitirán realizar prácticamente todo los que nos venga a la cabeza.

```
//Ejemplo de uso de los punteros
#include <stdio.h>

int main(){
   int a=3, b=8; //Declaramos dos variables a y b con valores 3 y 8
   int *z,*p;    //Declaramos dos punteros z y p
   z=&a          //Guardamos el nombre de la cajita de a en z
   p=z           //Copiamos el contenido de z en p

   /*En este punto tanto p como z están apuntando a la misma cajita que contiene
el valor de a */

   b=*p+3       //b ahora es igual al contenido de la cajia apuntada por p más 3
   printf(“%d%d”,a,b); //Finalmente, esta instrucción imprimirá  3 y 6
}
```

Con esto finaliza la pequeña introducción al infinito mundo de los apuntadores. Los invito a probar distintas instrucciones utilizando operadores hasta comprender completamente cómo se comportan antes de seguir avanzando.

Pasaje por referencia

Para finalizar el capítulo ya solo nos resta hablar de cómo funciona el pasaje por referencia.

Ya dijimos que cuando una variable es declarada dentro de una función esta será destruida al finalizar la llamada a la función, y nuestra única manera de conservar el valor de alguna de ellas es por medio de la instrucción **return**. Sin embargo, ¿qué sucede cuando necesitamos conservar más de un valor durante una llamada a una función?. La solución más obvia es crear variables globales, pero esto no siempre es conveniente. La segunda opción es utilizar pasaje por referencia.

El pasaje por referencia ocurre cuando en vez de pasar el valor de una variable como parámetro, lo que le damos a la función es la dirección de memoria que almacena a dicho valor. De esta manera, todos los cambios ocurrirán directamente en la cajita de memoria correspondiente a la variable local del ambiente que realizó la llamada a la función, conservando así todos los cambios realizados a las variables pasadas por referencia.

```
//Ejemplo de definición de función con
//pasaje de parámetros por referencia

void función (int *a, int *b);
```

Este último tema puede resultar infinitamente confuso; es por esto que recomiendo leer el último párrafo una y otra vez hasta que se comprenda cada oración, y luego practicar el comportamiento de una función específica para ambos tipos de pasaje de parámetros.

```
//Función que toma 2 variables e intercambia sus valores
#include <stdio.h>

void Intercambio (int *a, int *b); //Definición de la función

int main ()
{
   int num1 = 4, num2 = 6;
   Intercambio(&num1,&num2); //Notemos que no hace falta declarar punteros
                             //Basta con pasar la dirección de memoria y
                             //esta se almacenará en el puntero que
                             //declaramos como parámetro
}

void Intercambio (int *a, int *b) //Declaración de la función
{
   int aux;
   aux=*a;
   *a=*b;
   *b=aux;
}
```

Tipos de Datos Estructurados

De manera muy resumida, los tipos de datos estructurados son aquellos formados por la agrupación de tipos de datos simples; como lo son **int**, **float**, y **char.** Los tipos de datos estructurados más comunes son los vectores, matrices, cadenas de caracteres y estructuras. En lo que resta de la guía hablaremos de estos tipos y como utilizarlos.

Arreglos (Vectores)

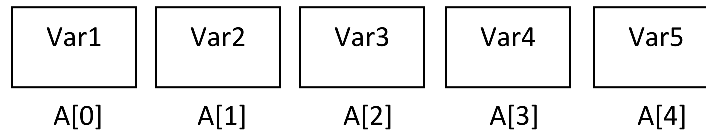
Los arreglos son el tipo de datos estructurados más sencillo que existe. Los arreglos nos permiten agrupar datos del mismo tipo básico bajo un mismo identificador. Cada arreglo tendrá un tamaño definido fijo, el cual se determina durante la declaración; este tamaño es el que determinará cuantas variables contendrá nuestro arreglo.

```
//Sintaxis de la declaración de un arreglo

[Tipo] <Nombre> [<número de elementos>];
```

Lo primero que notamos en la sintaxis es la necesidad de un tipo, este determinará el tipo de datos que se podrán almacenar dentro del arreglo. Lo siguiente es colocar nombre a nuestro arreglo, siguiendo las mismas reglas que utilizamos para variables y funciones. Los corchetes en negritas forman parte de la sintaxis, dentro de ellos colocaremos un numero natural, el cual nos indicará el tamaño del arreglo; es decir, el número de elementos que compondrán nuestro vector.

```
//Ejemplo de declaración de un arreglo

int A[20];
```

Para acceder a los distintos datos almacenados dentro de nuestro arreglo se hace uso de un subíndice, el cual indicará la posición del dato al que queremos acceder. Las posiciones en los arreglos van desde cero (siendo este el primer dato) hasta n-1 (siendo n el tamaño del arreglo). El error más común al momento de trabajar con arreglos es olvidar que el índice del primer elemento es [0] y no [1]. Por ejemplo, si declaráramos un arreglo de 5 posiciones llamado A, gráficamente se vería de esta manera.




```
//Ejemplo de declaración de un arreglo y acceso a los datos en el mismo

int A[5] = {1,2,3,4,5}; //Inicialización de un arreglo durante su declaración
                        //Si no le colocamos tamaño al arreglo este será del
                        //tamaño de valores que le asignemos

int B[5];

B[0] = 1; //Inicialización de un arreglo luego de su declaración
B[1] = 2;
B[2] = 3;
B[3] = 4;
B[4] = 5;
```

```
//Ejemplo de Asignación de valores

int A[10];
int i;

for(i=0;i<10;i+=1) A[i]=0;
```

```
//Ejemplo de lectura de valores

int V[10];
int i;

for(i=0;i<10;i+=1){
  printf("introduce V[%d]",i);
  scanf("%d",&V[i]);
}
```

Arreglos Bidimensionales (Matrices)

Cuando declaramos un arreglo no solo debemos definir su tamaño, sino también su dimensión. La dimensión de un arreglo dependerá del número de corchetes que le coloquemos luego del nombre.

```
//Arreglos de distintas dimensiones

int A[10];          //Arreglo Unidimensional (Vector)
int B[10][5];       //Arreglo Bidimensional (Matriz)
char C[10][5][10]; //Arreglo Tridimensional
```

Un arreglo bidimensional se comporta de la misma manera que una matriz; en donde los subíndices serán la fila y la columna a la que queramos acceder. El siguiente ejemplo ilustra gráficamente un arreglo bidimensional declarado como “int A[3][5]”.

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]

```
//Ejemplo de Asignación de valores

int A[3][5];
int i,j;

for(i=0;i<3;i+=1)
  for(j=0;j<5;j+=1) A[i][j]=0;
```

Cadena de Caracteres (Strings)

Las cadenas de caracteres son casos especiales de arreglos unidimensionales, en los que el tipo de datos recibido es **char**. C tiene un tratamiento especial para estos arreglos a los cuales denominamos **Strings**.

Quizá la consideración más importante a tener cuando trabajamos con Strings es la siguiente: C inserta al final de cada cadena de caracteres un cero, indicando el final de la misma; de esta manera podemos tener frases de menor tamaño al del arreglo y C siempre sabrá donde terminar. Veamos los ejemplos.

```
//Manejo de Strings en C

char Nombre[10] = “maria”; //Declaramos una un string de tamaño 10, el cual
                           //puede almacenar frases de hasta 9 caracteres
                           //siendo 0 el decimo carácter.

char Nombre[] = “maria”;    //En este caso, el string tendrá tamaño 6, las
                           //5 letras de “maria” más el carácter nulo

char Nombre[10] = {‘m’,’a’,’r’,’i’,’a’,’\0’}; //Aquí declaramos un string
                                              //tamaño 10 y especificamos
                                              //directamente el contenido

//Nótese que en el último caso es necesario incluir manualmente el 0
//Utilizando el carácter de escape “\”

/* Lectura y escritura */

printf(“%s”,Nombre);
scanf(“%s”,Nombre);
```

Estructuras

Finalmente llegamos al tipo de datos estructurados más complejo. Las estructuras nos permitirán unir no solo los tipos de datos simples, sino también arreglos multidimensionales y hasta otras estructuras. Cada estructura que definamos puede tener un número infinito de variables internas, cada una de las cuales puede ser de un tipo distinto.

A continuación se listarán cada una de las características de las estructuras, presentando un ejemplo para cada una. Se recomienda practicar cada una y asegurarse de comprenderla completamente antes de seguir avanzando.

```
//Sintaxis de declaración de una estructura

struct <Identificador>
{
  [tipo] <nombre_objeto>;
  [tipo] <nombre_objeto>;
  [tipo] <nombre_objeto>;
  ...
} <objeto_estructura>;

/* Nota: Tanto el identificador como el objeto estructura son opcionales,
pero al menos uno de los dos debe existir */
```

Luego de declarada la estructura, si hemos especificado un nombre para ella, se puede utilizar como cualquier otro tipo.

```
//Declaración de un objeto estructura

struct <identificador> <objeto_estructura>;
```

Para acceder a un objeto interno de una struct usamos el operador "."

```
//Ejemplo de acceso a una estructura

struct persona
{
  char Nombre[10];
  char dirección [20];
  int anoNacimiento;

} fulanito;

fulanito.anoNacimiento = 1991;

fulanito.Nombre = "Fulano";
```

La asignación entre estructuras está permitida solo si son del mismo tipo.

```
//Asignación entre estructuras

struct persona fulano;

fulano = fulanito;
```

Finalmente, la inicialización de las estructuras se realiza encerrando entre llaves las inicializaciones individuales de cada tipo que encierra una estructura partículas, separándolas con coma. Se debe tener cuidado con las estructuras anidadas; cada nueva estructura anidada deberá iniciarse usando la pareja correspondiente de llaves tantas veces como sea necesario.

```
//Inicialización de estructuras

struct A{
  int x;
  struct c{
    char c;
    char d;
  }y;
  int z;
};

struct A ejemploA = {10,{‘a’,’b’},20};
```