Capítulo 1

Instrucciones del Lenguaje C

Puesto que C es un lenguaje imperativo, un programa está conformado básicamente por una secuencia de *órdenes* que llamaremos comandos o instrucciones. En este capítulo presentaremos las instrucciones básicas del lenguaje C y mostraremos como construir programas que las usan.

Distinguiremos dos tipos de instrucciones que llamaremos *instrucción simple* e *instrucción compuesta* respectivamente. La sintaxis de una instrucción simple es una expresión seguida de un punto y coma (;). Mientras la sintaxis de una instrucción compuesta es una llave que abre ({) seguida de una secuencia de instrucciones simples o compuestas, seguida de una llave que cierra (}). Todas las intrucciones que presentaremos son simples salvo una que llamaremos **bloque**.

1.1. Declaración de Constantes y Variables

Una declaración es una orden que tiene como finalidad solicitarle al sistema operativo que reserve un espacio de memoria adecuado para almacenar el valor de un dato del que se especifica el tipo y el nombre. El tipo se especifica usando una palabra reservada para denotar un tipo básico, como por ejemplo *char, int, float* o el nombre de un tipo previamente definido por el usuario, mientras que el nombre se especifica usando un identificador. La sintaxis de la declaración es la siguiente:

```
tipo <identificador>;
tipo <identificador>, ..., <identificador>;
const tipo <identificador>;
const tipo <identificador>, ..., <identificador>;
```

Note que por ser una instrucción simple la expresión de la declaración debe terminar en (;). La primera declara una variable de tipo "tipo", la segunda declara una secuencia finita de variables de tipo "tipo". Mientras que las últimas dos usan la palabra reservada **const**, como **modifcador** para indicarle al sistema operativo que, una vez que la posición de memoria que se reserva se inicialice, no

se le puede cambiar el valor. Esto es, son constantes. Es importante entender que al declarar las variables sólo se reserva el espacio de memoria; dicho espacio de memoria puede contener cualquier secuencia de bits que que será interpretada como el valor correspondiente al tipo de variable que hemos declarado—puesto que su valor no significa nada, lo interpretamos como si su contenido es basura. Para darle un valor a una variable recién declarada hay que usar otras instrucciones que veremos luego.

A continuación se muestra varios ejemplos de declaración de variables.

```
char c;
int m, n;
float x, y, z;
const int A, B;
```

La primera declara una variable de tipo carácter de nombre c, la segunda dos variables enteras de nombres m y n, la tercera tres variables punto flotantes de nombres x, y, z y la últina declara dos constantes enteras de nombres A y B.

A parte de las variables de los tipos básicos: *char, short, long, int, etc.* se pueden declarar variables de otros tipos como, por ejemplo **apuntadores**, que usan para almacenar la posición de memoria de una variable de algún tipo y arreglos que sirven para almacenar bajo el mismo nombre varias variables de un mismo tipo.

Ejemplo 1.1 *Declare dos arreglos de enteros de nombres a y b de tamaño* 10 *y dos apuntadores a enteros de nombres p y pr.*

Explicación: Para declarar arreglos se coloca después del nombre de la variable un corchete que abre, el tamaño del arreglo y un corchete que cierra. en nuestro caso queda.

```
int a[10], b10];
```

Para declarar un apuntador a un entero se coloca antes del nombre de la variable un *asterisco*, en nuestro caso queda:

```
int *p, *pr;
```

Nota: una declaración del estilo int a[n]; es incorrecta a menos que la variable n se haya declarado e inicializado con anterioridad. •

Este ejercicio muestra que el tipo de declaraciones que se pueden hacer es realmente más amplio que el sugerido por nuestra presentación inicial.

1.2. Asignación

La instrucción de asignación consiste en cambiar el valor de una variable por otro o darle a una constante un valor que luego no podrá modificarse. Para ello el lenguaje C utiliza el símbolo (carácter) (=). Su sintaxis es como sigue:

```
<identificador> = <expresion>;
<identificador> = <expresion>, ..., <identificador> = <expresion>;
```

<identificador>es un identificador que ha sido declarado con anterioridad y <expresion>es una expresión que debe ser del mismo tipo que se usó para declarar al identificador.

Ejemplos...

```
c = c + 'A', d = '\a';
n = 2*(a<b ? a : b);
m = 2*n - 1;
x = 1.0, z = y/(x+1);
A = 18, B = m%4;
```

1.3. Declaración y Asignación Combinadas

En algunas ocasiones es conveniente combinar las intruciones de declaración y de declaración en una sola. Esto se logra de la sigiente manera:

```
tipo <identificador> = <expresion>;
tipo <identificador> = <expresion>, ..., <identificador> = <expresion>;
const tipo <identificador> = <expresion>;
const tipo <identificador> = <expresion>, ..., <identificador> = <expresion>;
```

La parte correspondiente a la asignación es opcional, esto es, en cualquiera de estas lista, en cualquiera de las sub-expresiones <identificador>= <expresion>se puede omitir la parte = <expresion>. Es importante resaltar que el tipo de la expresión debe coincidir con el tipo de la variable que se declara (esto es, con "tipo").

Veamos algunos ejemplos.

```
char c = 'A', d = '\n', e;
int m, n = 8;
float x = .0, y = 3.1415, z;
const int A = 18, B;
```

1.4. Asignaciones Especiales

Puesto que con mucha frecuencia se incrementa o se decrementa el valor de una variable en una unidad, el lenguaje C proporciona una forma más compacta y más eficiente de hacer estas operaciones, mediante los siguiebtes cuatro expresiones que mostramos a continuación usando la variable k:

k++, ++k, k-- y --k

Esta variable no tiene que ser entera, puede ser de tipo punto flotante o de tipo carácter, incluso de ciertos tipos que aún no hemos discutido. Recuerde que cada carácter es a su vez un número y representan (forman) una secuencia ordenada. Estas expresiones, en algún sentido implementan el tipo sucesor y predecesor.
Estas expresiones tienen la peculiaridad de poder usarse por si mismo como una instrucción si se las escribe seguida de un (;) o de poder ser usadas como expresion o sub-expresión. La interpretación operacional de ++k es la siguiente: "si se requiere usar el valor que k almacena se incrementa en uno y luego se usa". Mientras que la interpretación operacional de k++ es "si se requiere usar el valor que k almacena se usa y luego se incrementa en uno". En otras palabras, ++k representa el núevo valor de k mientras que k++ representa el actual. Así, si k vale 8, la asignación $k1 = ++, k2 = ++k$ hace que k1 tenga el valor 8 y k2 tenga el valor 10. El comportamiento de k– y –k es análogo.

A continuación mostramos unos ejemplos de uso:

Ejemplo 1.2 *Algunos ejemplos:*

```
k++;
++k;
r = r*++k;
s = k*k++, r = k*++k;
```

Explicación: Los primeros dos tienen el mismo efecto pues el valor de k no se esta usando—sólo se hace el incremento. el tercero asigna a r el valor de r multiplicado por $k+1$, esto es equivalente a $r = r*(k+1), k = k+1$; o a $r = r*(k+1), k++$;. La última es equivalente a $s = k*k, k = k+1, r = k*(k+1), k = k+1$;. •

El lenguaje C también provee una forma compacta y eficiente para los acumuladores mediante varias variantes de la asignación. La sintaxis es

<identificador><op>= <expresion>;

Donde <identificador>es el nombre de una variable, operador es uno de los siguientes operadores binarios $+, -, *, /, \%$ y <expresion>es una expresión del mismo tipo que la variable. La interpretación operacional es la siguiente: " a la variable de nombre <identificador>y el valor de la expresión <expresion>se operan con el operador binario <op>y el resultado se almacena en la variable de nombre <identificador>". Esto es, es equivalente a

<identificador>= <identificador><op><expresion>;

Ejemplo 1.3

Algunos ejemplos:

```
sum += 2*k + 1;
r %= k+1;
```

Explicación: La primera incrementa el valor de la variable *sum* en el valor de la expresión $2 * k + 1$; es equivalente a "sum = sum + 2*k + 1;". La segunda coloca en r el resto de dividir a r entre $k + 1$; es equivalente a "r = r •

1.5. Invocación de Sub-programas

Un sub-programa es un programa que realiza una tarea específica y que puede usarse dentro de otro programa. Formalmente hay varios tipos de sub-programas que no discutiremos aquí, al menos, por ahora. Pero es importante aclarar que algunos de ellos se pueden usar como instrucciones simples mientras que otros se pueden usar como expresiones. Hay varios tipos sub-programas: a los que después de su ejecución retornan un valor calculado en base a los datos de entrada y no modifican el espacio de estados del programa que los invocó se llaman funciones. A los que no retornan un valor pero pueden modificar el espacio de estados se les llama procedimientos. Hay otros que son hibridos entre estos dos casos, etc. Aquí, por ahora, estamos interesados sólo en los procedimientos pues al ivocar un procedimiento basta con poner un punto y coma al final de la invocación para tener una instrucción. No ocurre lo mismo al invocar una función.

Lo ilustraremos mostrando cómo se invocan dos de los sub-programas incluidos en la librería *stdio.h*: **printf** y **scanf**.

1.5.1. Sub-programa printf

El sub-programa **printf** permite imprimir texto y valores de variables. Se invoca con la siguiente sintaxis:

```
printf(<arg0>, <arg1>, ... ,<argk>);
```

donde <arg0>es una cadena de carácteres que se anota entre comillas dobles, como por ejemplo: `"Hola mundo\n"` o `"El número de divisores de %d es %d."` La primera sólo contiene carácteres, mientra que la segunda contine dos "directivas" que comienzan con el carácter %—luego indicaremos para que sirven. El resto de los argumentos <arg1><argk>son expresiones y deben ser igual, en número, al número de directivas `%`, presentes en <arg0>.

`printf(<arg0>, <arg1>, ... ,<argk>);`, imprime la cadena de carácteres <arg0>pero con las directivas `%?` sustituidas por los valores de las variables dadas en la lista de argumentos <arg1>, ..., <argk>en el orden en que aparecen.

Ejemplo 1.4 *¿Qué imprimen las siguientes invocaciones del comnado* **printf***?*

```
printf("Hola mundo\n");
int n= 12, d = 6;
printf("El numero de divisres de %d es %d.", n, d);
```

Explicación: La primera escribe: Hola mundo, pero además salta a la siguiente línea debido al carácter `\n`. Mientras que la segunda imprime: El numero de divisores de 12 es 6., pues sustituye el primer `%d` por el valor de la varlor de la expresión n, que es 12 y al segundo `%d` por el valor de la expresión d, que es 6. •

1.5.2. Sub-programa scanf

El sub-programa **scanf** permite leer datos básicos en variables de variables. Se invoca con la siguiente sintaxis:

```
scanf(<arg0>, <arg1>, ... ,<argk>);
```

donde <arg0>es una cadena de carácteres que se anota entre comillas dobles, que contiene especificadores de conversión, el resto de los argumentos <arg1>, ..., <argk>son posiciones de memoria donde se almacenaran las conversiones hechas siguiendo las indicaciones en <arg0>.

Ejemplo 1.5 *¿Qué hacen las siguientes invocaciones de* **scanf***?*

```
scanf("%d%d",&a, &b);
scanf("%f%d",&x, &y);
```

Explicación: La primera busca en el flujo de entrada dos enteros y los almacena en las posiciones de memoria que se indican. Realmente busca en la cadena de carácteres una secuencia de caracteres que se pueda convertir en un entero, lo convierte y lo almacena. •

Un error muy frecuente al usar el comando **scanf** es no usar el operador de indireccionamiento `&` que devuelve la dirección de memoria de la variable que le sigue. Se suele escribir `scanf("%d", m);` en lugar de `scanf("%d", &m);`. Por lo general, el compilar no detecta este error y en la ejecución el programa no lee el valor que se le indica. Puesto que **scanf** devuelve el número de argumentos leidos se puede usar este valor para saber si, en efecto, la lectura fué satisfactoria.

1.6. El Comando Return

El comando o instrucción **return** se usa para terminar la ejecución de un programa en cualquier punto de la misma y devolver el valor que se espera produzca como salida dicho programa. Su sintaxis es:

```
return <expresion> ;
```

Donde <expresion>es una expresion del tipo que se espera devuelva el programa que la usa. Lo más elegante es que haya un solo **return** y que sea la última instrucción del programa. Pero hay situaciones donde conviene usarlo en medio del código. Alerta rojo: en un programa correcto cualquier ruta que siga la ejecución de un programa debe finalizar con un **return**. A menos que se declare como **void** , todo sub-programa debe terminar con un return.

Ejemplo 1.6

```
int min(int a, int b){
   return a<b ? a : b;
}
```

Explicación: Este programa sólo usa el comando **return** para retornar el valor del mínimo entre a y b. •

1.7. Bloques

Un bloque es una secuencia de *instrucciones* que conforman una **instrucción compuesta.** Para delimitar las instruciones que forman un bloque se usan las llaves { }. Esto es, se encierra la secuencia de instruciones entre llaves. Después de la llave que termina el bloque no se necesita escribir un *punto y coma* (;). Un punto muy importante es que las variables definidas dentro de un bloque son *locales*, esto es, desaparecen al terminar el mismo—su vida comienza en el punto en que se las declaró y termina al terminar el bloque. Si dentro de un bloque se declara una variable con el mismo nombre que una declarada fuera del bloque, al hacer mención de dicha variable se usa la local. Las definidas antes del bloque son globales, esto es, se pueden usar tanto fuera como dentro del bloque, a menos que se hayan redefinido.

Un bloque se puede crear en cualquier punto de un programa pero son particularmente útiles como cuerpo de un sub-programa—por ejemplo, el **main**, o de instrucciones que veremos más adelante como la instrucción condicional y la de repetición. Veamos algunos ejemplos:

Ejemplo 1.7

```
int a = 5, b = 3;
{ int aux = a; a = b; b = aux;}
```

Explicación: Estas dos lineas de código están formadas por dos **instrucciones**, una simple que declara dos variables y las inicializa, y una compuesta: el bloque, que intercambia sus valores usando una variable auxiliar que desaparece al terminar la instrucción. Note que si tratara de usar el valor de *aux* fuera del bloque el programa daría un error. •

Ejemplo 1.8

```
void main() {
   printf("Hola mundo!!!");
}
```

Explicación: El cuerpo de este programa es un bloque. De hecho, el cuerpo de un sub-programa en C siempre es un bloque. •

Ejemplo 1.9 *¿Cuál es la salida del siguiente del siguiente pedazo de código?*

```
{   int a = 11, b =5;
    {  int a = 5, x = b;
       b = a+b; a /= 2;
       printf("a = %i, b = %i, x = %d.\n", a, b, x);
       int b = 7 - 2*a;
       x += b, a = a*b;
       printf("a = %i, b = %i, x = %d.\n", a, b, x);
    }
   printf("a = %d, b = %d.\n", a,b);
}
```

Explicación: En el primer printf la variable a debe valer 2, pues es la a del bloque interno —se redefinió la variable—se le asignó el valor 5 y se dividió entre 2. Mientras que el valor de b debe ser 10—suma del valor de la variable a del bloque interno con la b del blque externo. Claramente el valor de la x es 5, no? La salida completa es:

```
a = 2, b = 10, x = 5.
a = 6, b = 3, x = 8.
a = 11, b = 10.
```

•

1.8. Comando No-Hacer-Nada

El comando *no-hacer-nada* se puede representar en C de dos maneras, a saber: Puesto que las instrucciones simples en C sintacticamente se representan como una expresión seguida de un *punto y coma*(;), si asociamos con la instrucción *no-hacer-nada* a la palabra vacía—o si se quiere a la expresión vacía, entenderemos el porqué en C la instrucción *no-hacer-nada* se representa como el *punto y coma*(;). Otra forma de representar a la instrucción no-hacer-nada es por medio de un bloque vacío, esto es, un bloque sin instruciones—un bloque con una secuencia vacía de instruciones. Note que éstas son justo dos, formas de representar a la instrucción *no-hacer-nada* pues cualquier combinación de ellas también la representan. El nombre en inglés para esta instrucción es **skip**.

Como consecuencia de esta interpretación del (;), se puede escribir, por ejemplo, ";"después de un bloque aunque no es necesario; simplemente se está agregando una instrucción **skip**; o se pude escribir cualquier secuencia de puntos y coma, como por ejemplo: ;;;.

1.9. Instrucción condicional

Una instrucción condicional o condicionada, es una instrucción que se ejecuta si y sólo si se satisface una condición pre-establecida. Su sintaxis en el lenguaje C es:

```
if (condición) Instrucion1
```

Su interpretación operacional es: *Se evalúa la condición, si resulta verdadera, se ejecuta la instrucción: Instrucción1, en caso contrario no se hace nada.*

Veamos algunos ejemplos:

Ejemplo 1.10

```
if (a <= b) min = a;
if (a <= b) {min = a; max = b;}
if (b != 0) {q = a /b; r = a % b;}
```

Explicación: En el primer ejemplo si la condición se satisface se ejecuta una uínica instrucción, a saber: en min se copia el valor de a. Puesto que es una sóla enstrucción, no se requieren las llaves, pero no hace daño si se escriben. En el segundo ejemplo, si la condición se cumple, se ejecutan las dos instrucciones dentro de las llaves. Note que cada instrucción termina con *punto y coma*, pero que despues de la llave que cierra no se requiere el *punto y coma*. Note también que es distinto anotar:

```
if (a <= b) min = a; max = b;
```

¿Por qué? El tercer ejemplo es similar al segundo; se determinan el cociente y el resto, sólo si si $b \neq 0$. •

Ejemplo 1.11 *Escriba un programa que permita leer tres enteros a, b, c y dé como salida un entero min que sea el mínimo de dichos enteros.*

Explicación: Nuestra estrategia consistirá en considerar que el primer número es el mínimo, luego compararlo con el siguiente y si el siguiente es más pequeño, entonces él será el mínimo, y finalmente compararemos el tercer número con el mínimo, y si él es más pequeño, el será el mínimo.

Para ello empezamos declarando y leyendo las variables enteras a, b, c. Luego, declaramos la variable de salida *min* e inicializandola—por defecto—en a. Aquí, necesitamos una comparar la variable b con el mínimo actual para actualizar el mínimo, si hiciese falta. Finalmente comparamos c contra el mínimo y actualizamos si hiciera falta. El código en c es:

```
int main(){
   int a, b, c;
   printf("Dime tres enteros: ");
   scanf("%d%d%d", &a, &b, &c);
   int min = a;
   if (b < min) min = b;
   if (c < min) min = c;
   printf("\nEl minimo entre %d, %d y %d es: %d.\n", a, b, c, min);
   return 0;
}
```

Este código es optimal, esto significa que no se puede hacer menos de las comparaciones que él hace para determinar el mínimo. •

1.10. Bifurcación o Alternación

La instrucción de alternación o bifurcación permite bifurcar el flujo de un programa en base a un perdicado sobre las variable del espacio de estados. Su sintaxis es la siguiente:

```
if (condición) Instrucion1 else Instruccion2
```

Su semántica es la siguiente: se evalúa la condición ..., si resulta verdadera se ejecuta la instrucción: Instruccion1, si no se ejecuta la instrucción: instruccio2. Recordemos que tanto Instrucio1 como Instrucción2 pueden ser simples o pueden ser un bloque. A continuación mostramos un programa sensillo que usa una instrucción de bifurcación.

Ejemplo 1.12 *Escribir un programa que permita decidir si tres puntos del plano son o no colineales.*

Explicación: Tres puntos del plano son colineales ssi el tercer punto pertenece—satisface la ecuación— a recta formada por los dos primeros. Puesto que la ecuación de la recta que pasa por el punto (x_0, y_0) y tiene pendiente m viene dada por $y - y_0 = m(x = x_0)$ y la pendiente de la recta que pasa por los puntos (x_0, y_0) y (x_1, y_1) es $\frac{y_0 - y_1}{x_0 - x_1}$ se tiene que la ecuación de la recta, escrita sin denominadores para evitar el problema de división por cero, es (x1-x0)*(y-y0) = (y1-y0)*(x-x0). Luego, los tres puntos son colineales si se cumple la siguiente condición: $(x1 - x0) * (y2 - y0) == (y1 - y0) * (x2 - x0)$ y el siguiente programa da una solución al problema.

```
void main(){
   int x0, x1, x2, y0, y1, y2;
   printf("Puntos ");
   scanf("%d%d%d%d%d%d", &x0, &y0, &x1, &y1, &x2, &y2);
   if ((x1-x0)*(y2-y0)==(y1-y0)*(x2-x0)) printf("\nSon colineales");
   else printf("\nNo son colineales");
}
```

•

Ejemplo 1.13 *Escriba un programa que permita determinar el máximo y el mínimo de un conjunto de tres enteros a, b, c.*

Explicación: La estrategia será usar la instrucción de bifurcación para determinar con una sola comparación el mínimo y el máximo entre los primeros dos enteros y luego usaremos otra vez la instrucción de bifurcación para derterminar si c es el mínimo o el máximo.

```
int main(){
   int a, b, c;
   printf("Dime tres enteros: ");
   scanf("%d%d%d", &a, &b, &c);
   int min, max;
   if (a < b) {min = a, max = b} else {min = b, max =a;}
   if (c < min) min = c; else if (c > max) max =c;
   printf("\nMin y Max entre %d, %d y %d son: %d y %d.", a, b, c, min, max);
   return 0;
}
```

Observe que la acción del else del if es a su vez una acción condicionada. Esta es una estrutura muy frecuente y a su vez muy útil. Esta en el peor de los casos sólo hace una asignación. En el peor caso hace dos comparaciones, pero en el mejor de los casos sólo hace una. Si la cambiara por dos instrucciones condicionales separadas como:

```
   if (c < min) min = c;
   if (c > max) max =c;
```

siempre se harían dos comparaciones y una sola asignación. •

Nótese que si tomamos a Instrucción2 como a la instrucción **skip** la instrucción `if (condición) Instrucion1 else Instruccion2` es equivalente a `if (condición) Instrucion1 else ;`

Ejercicio 1.1 *Escriba un ejemplo que ponga en evidencia que las siguientes dos instrucciones no son equivalentes a la instrucción* `if (expBool) I1 else I2`

```
if (expBool) I1
if (!expBool) I2
```

1.11. El Comando Switch

La instrucción **switch** es una instrucción que permite chequear si una expresión entera dada coincide con una de varias constantes enteras definidas con anterioridad para bifurcar de acurdo con el valor en que coincidan. Su sintaxis básica es:

switch (<expresion>) {
  case expr-const : *instrucciones*
  ⋮
  case expr-const : *instrucciones*
  default : *instrucciones*
}

Una expresión constante debe aparecer sólo una vez y en cualquier orden. La etiqueta *default* es opcional. Su interpretación operacional es:"se evalúa la expresión <expresion>, si la misma coincide con alguna de las 'éxpr-const', se empieza la ejecusión con las *intrucciones* correspondientes a dicho caso y se continúa con las de los casos siguientes, a menor que se interrumpa la instrucción **switch** con el comando **break**; si no coincide con ninguna se hacen las *instrucciones* del *default*—si lo hay." Salvo casos muy particulares conviene que la última instrucción de cada grupo de *instrucciones* se **break;**. Por ejemplo, si para varios casos se deben ejecutar las mismas intruciones, en lugar de repetirlas en cada uno de esos caso se ...

switch (<expresion>) {
  case 4 : ;
  case 2 : ;
  case 5 : printf("Manejo casos 4, 2, 5"); ... break;
  case 1 : ;
  case 3 : printf("Manejo casos 1, 3"); ... break;
  case 6 : printf("Manejo 6"); ... break;
  default : printf("Manejo el resto de los casos"); ... break;
}

Si el valor de <expresion>fuese 4, se ejecutan las instrucciones de caso 4, que *no-hacer-nada*, las de caso 2, que de nuevo es *no-hacer-nada* y luego las del caso 5 que terminan con un **break** que interrumpe la ejecución. Si el valor de <expresion>fuese 2, se hace un **skip** y las de caso 5. Casos 1 y 3 se manejan de manera similar.

Ejercicio 1.2 *Escriba un programa en C que reciba como entrada un número entero positivo mes menor que 13, que representa un més del año y dé como salida un entero de nombre dia que representa el número de días del més mes y un mesajes de estilo, en una linea propia: "El mes 10 tiene 31 dias."*

1.12. El Comando goto

La instrucción o el comando **goto**, que significa en español **ir-a**, tiene como función cambiar el flujo normal del programa. Su sintaxis es

```
goto <itiqueta>;
```

donde <itiqueta>no es más que un identificador que se coloca delante de una instrucción. A continuación mostramos un ejemplo:

Ejemplo 1.14 *Escribamos un programa que sume los primeros n enteros positivos.*

Explicación: Se nos pide hacer la siguiente suma: $\sum_{k=1}^{n} k = 1 + 2 \cdots + n$. Aparte de n que la declaramos constante necesitamos dos variables: una para acumular las sumas parciales que llamaremos s y una para llevar el sumando actual—indice de la suma—que llamaremos k. A la primera se le suele llamar *acumulador* y a la segunda, *índice o contador*. A la primera la inicializaremos en CERO, pues la suma de cero números es CERO y a la segunda en UNO obedeciendo al índice de la suma arriba anotada.
Le colocamos una etiqueta a la primera instrucción (ini :) ella sólo sirve para identificar a la línea. Si k fuera mayor que n ya habría terminado de sumar, no? Luego, le ordenamos al programa ir a la nínea identificada por la etiqueta (fin :) a iprimir el resultado. DE lo contrario, el **if** se comporta como un **skip** y se pasa a hacer la siguientes instrucciones: actualizar s, actualizar k e ir a (ini); y magicamente, estamos repitiendo el proceso.

```
void main() {
   const int n = 5;
   int k = 1, s = 0;
   ini : if (k > n) goto fin;  // o ini : if (k<=n) ; else goto fin;
   s = s + k;
   k = k + 1;
   goto ini;
   fin : printf("\nLos primeros %d enteros positivos suman %d.",n,s);
}
```

Note que pudimos haber seguido la siguiente suma $\sum_{k=0}^{n-1} k + 1 = 1 + 2 \cdots + n$ nos hubiera quedado el siguiente código. Sólo cambian tres líneas!

```
void main() {
   const int n = 5;
   int k = 0, s = 0;
   ini : if (k >= n) goto fin;  // o ini : if (k<n) ; else goto fin;
   s = s + k +1;
   k = k + 1;
   goto ini;
   fin : printf("\nLos primeros %d enteros positivos suman %d.",n,s);
}
```

•

Es muy importante resaltar que en la actualidad es casi como cometer un pecado usar la instrucción goto. La misma fue fuertemente criticada y casi condenada a muerte por que en manos poco expertas puede producir programas

muy difíciles de entender y de darles mantenimiento y lo más importante muy difíciles de probar su crrección. Se recomienda que se use o que se use sólo en casos muy especiales.
Sin embargo, creemos que entender su funcionamiento, puede aclarar considerablemente el funcionamiento de los ciclos de repetición.

Ejercicio 1.3 *Usar la instrucción* **goto** *para escribir un programa en C que permita hallar la suma de los primeros n múltiplos de tres.*

Ejercicio 1.4 (Factorial) *Escriba un programa en C que use la instrucción* **goto** *y que permita hallar el factorial de un número natural n. Recuerde que el factorial de un número se define recursivamente como sigue:*

$$n! = \begin{cases} 1, & si\ n = 0; \\ n(n-1)!, & si\ n > 0. \end{cases}$$

Por ejemplo, $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24$

1.13. Ciclos de Repetición

1.13.1. La instrucción while

La instrución **while** permite repetir una instrucción mientras una condición **booleana** se mantenga **VERDADERA**. Su sintaxis es la siguiente:

```
while (exprBool) Instrucción
```

Su semantica es la siguiente: se evalúa la expresión booleana *exprBool*, si resulta VERDADERA se ejecuta la instrucción *instrucción* y se repite la intrucción *while*, esto es, se evalua la expresión booleana *exprBool*, etc.; en caso contrario se da por terminada la ejecución de la instrucción *while*. en otras palabras: "Mientras la expresión booleana sea cierta se ejecuta la instrución". Nótese que es una instrución condicionada que se repite si la condición es VERDADERA.

Repitamos el ejemplo de calcular la suma de los primeros enteros positivos usando la instrución *while*.

Ejemplo 1.15 (Suma de Gauss) *Escribir un programa en C que use la instrucción* **while** *para hallar la suma de los primeros n enteros positivos.*

Explicación: Se nos pide calcular la siguiente suma: $S_n = \sum_{k=1}^{n} k = 1+2\cdots+n$. Aparte de n que la declaramos constante necesitamos dos variables: una para acumular las sumas parciales que llamaremos s y una para llevar el sumando actual—indice de la suma—que llamaremos k. A la primera se le suele llamar *acumulador* y a la segunda, *índice o contador*. A la primera la inicializaremos en CERO, pues la suma de cero números es CERO y a la segunda en UNO obedeciendo al índice de la suma arriba anotada.

```
void main() {
   const int n = 5;
   int k = 1, s = 0;
   while (k<= n) {
      s = s + k;
      k = k + 1;
   }
   printf("\nLos primeros %d enteros positivos suman %d.",n,s);
}
```

Nótese que el valor de n se le pudo haber solicitado al usuario. Esta suma tiene una fórmula cerrada y se le conoce como suma de Gauss. Dicha fórmula es $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. Por supuesto que un programa que use la fórmula es mucho más eficiente; aquí el interés es mostrar cómo se construyen programas usando ciclos de repeticiones. •

Ejemplo 1.16 (Primalidad) *Un entero positivo es primo si y sólo si tiene exactamente dos divisores positivos. Escriba un programa en C que permita decidir si un entero es dado n es primo. La variable b debe valer 1 si el entero n es primo y 0 en caso contrario. Debe usar la a la varuable b para dar un mensaje que indique si n es primo o no.*

Explicación: Para que un entero positivo n sea primo no debe existir ningún entero positivo mayor que 1 y menor que n que lo divida, por lo que parece una estrategia razonable asumir que el número es primo y revisar uno por uno los números entre 1 y n sin incluir buscando un posible divisor; si se encuentra uno que lo divida, entonces el número no es primo, de lo contrario lo es. Luego, necesitamos una variable adicional k que se mueva sobre los candidatos a di visores.

```
void main() {
   const int n;
   printf("Dame un entero positivo mayor que uno: ");
   scanf("%d",&n);
   int b = 1, k = 2;
   while (k<n) {
      if (n % k == 0) {b =0;}  //si k divide a n, n no es primo
      k = k + 1;
   }
   if (b) printf("\n%d es primo.",n); else printf("\n%d no es primo.",n);
}
```

Este programa es correcto, pero es muy ineficiente por dos razones: la primera es que cuando consigue el primer divisor de n debería no continuar con la búsqueda. Por ejemplo, si el n es divisible por 2 la búsqueda debería detenerse la primera vez que entra al ciclo. Esto se logra forzando la salida de alguna manera:

motraremos tres formas de hacerlo. La primera es usar el comando **break** que termina el ciclo de ejecución. Basta con cambiar la instrucción **if**

```
if (n % k == 0) {b =0; break;}  //si k divide a n, n no es primo
```

La segunda es hacer que la condición del while deje de cumplirse, asignandole por ejemplo a k el valor de n. Para ello se cambia el **if** por:

```
if (n % k == 0) {b =0; k = n;}  //si k divide a n, n no es primo
```

Otra forma, un poco menos eficiente que las dos anteriores de mejorarlo siría agregar una condición adicional al **while**. Basta con cambiar el **while** por

```
   while (k<n && b) {
```

El segundo cambio que mejora drásticamente el eficiencia del algoritmo es observar que n no puede tener un divisor más grande que su mitad. Esto permite cambiar la condición del **while** por $k \leq n/2$. Además, si ya sabemos que 2 no divide a n, entonces basta hacer la busqueda hasta $k/3$ y así sucesivamente, podemos concluir que basta hacer la busqueda hasta $k \leq n/k$ o equivalentemente $k^2 \leq n$. Esta conclusión está respaldada por un teorema que establece que "si n es un entero compuesto, tiene un divisor primo menor o igual que la raiz cuadrada de n". Con estos cambios nuestro programa se transforma en:

```
void main() {
   const int n;
   printf("Dame un entero positivo mayor que uno: ");
   scanf("%d",&n);
   int b = 1, k = 2;
   while (k*k <= n) {
      if (n % k == 0) {b =0; break;}  //si k divide a n, n no es primo
      k = k + 1;
   }
   if (b) printf("\n%d es primo.",n); else printf("\n%d no es primo.",n);
}
```

Si tuvieramos a la mano una lista de los números primos podríamos un poco más. •

Ejemplo 1.17 *Escribir un programa en C que permita hallar la suma de los primeras n potencias naturales de 2.*

Explicación: Se nos pide evaluar la siguiente suma: $S_n = \sum_{k=0}^{n-1} 2^k = 1 + 2 + \cdots + 2^{n-1}$

```
void main() {
   const int n;
   printf("Dame un entero positivo: ");
```

```
   scanf("%d",&n);
   int k = 0, p = 1, s = 0;
   while (k<n) {
      s = s + p;
      p = 2*p;
      k = k + 1;
   }
   printf("\nEl valor de la suma es %d.", s);
}
```

•

Ejemplo 1.18 (Números de Fibonacci) *La sucesión de Fibonacci se define recursivamente de la siguiente manera:*

$$Fib_n = \begin{cases} 1, & si\ n = 0 \vee n = 1; \\ Fib_{n-1} + Fib_{n-2}, & si\ n > 1. \end{cases}$$

Escribir un programa en C que permita hallar el $n-$ésimo termino de la sucesión de Fibonacci.

Explicación: Se nos pide hallar Fib_n: Declaramos una variable k que se moverá sobre los naturales hasta llegar a n y una variable x que almacenará el valor de Fib(k)...Puesto que cuando quiero usar la fórmula recursiva para calcular el valor de $F(k)$ se requieren dos valores anteriores necesitamos declarar al menos una nueva variable que llamaremos y y que almacenará $Fib(k+1)$. ...

```
void main() {
   const int n;
   printf("Dame un entero no negativo: ");
   scanf("%d",&n);
   int k= 0, x = 1, y = 1;
   while (k<n) { //Invariante: Fib(k) = x, Fib(k+1) = y
      y = x + y;
      x = y - x;
      k = k + 1;
   }
   printf("\nFib(%d) = %d.", k, x);
}
```

•

Ejercicio 1.5 *Un natural es perfecto si y sólo si la suma de sus divisores es dos veces el número. Esto es equivcalente a decir que la suma de sus divisores menores que el número es igual al número. Escriba un programa que permita decidir si un número natural es perfecto.*

Ejercicio 1.6 *Escriba un programa en C que permita calcular la suma de los digitos de un número entero positivo.*

1.13.2. El Ciclo "do-while"

La instrucción **do while** es útil cuando queremos repetir un conjunto de instruciones pero sabems que dicho conjunto de instrucciones se debe ejecutar por lo menos una vez. Su sintaxis es:

```
do Intruccion while (<exprBool>);
```

Y su interpretación operacional es: "se ejecuta instrucción **Instruccion**, y luego se evalúa la expresión <exprBool>, si <exprBool>es cierta, se ejecuta de nuevo el while." Ella es equivalente a:

```
Instruccion
while (<exprBool>) Instruccion
```

Como ejemplo supongamos que necesitamos leer un entero positivo n, y queremos no seguir adelante hasta no estar seguros de que, en efevto, el número leido es positivo. Lo haremos en el siguiente ejercicio.

Ejemplo 1.19 (Repetición de Lectura) *Escriba un trozo de código en C que permita leer un entero positivo en la variable n.*

Explicación: Usaremos la instrucción **do while** pues debemos leer el número por lo menos una vez. A continuación copiamos el código necesario.

```
do {
   printf("Dame un entero positivo: ");
   scanf("%d",&n);
}while (n <= 0);
```

Esto pide un número que lee en n, si $n \leq 0$, vuelve a pedir el número, de lo contrario termina. Por supuesto que este código puede odificarse para leer varios datos a la vez. •

En el siguiente ejemplo muestra como repetir la ejecución de un pedazo de código al menos una vez.

Ejemplo 1.20 (Repetición de un Programa) *Escriba un trozo de código en C que permita leer dos enteros positivos a y b y calcule el máximo común divisor de dichos numero. Debe pedir los números hasta optener dos enteros positivos y debe repetir el proceso tantas veces como quiera el usuario.*

Explicación: Puesto queremos que el proceso se ejecute al menos una vez, haremos un ciclo de repetición **do wihile** con tres tareas básicas a saber: leer los números, calcular el máximo com;un divisor y preguntar si continuar. En seudo código luce así:

```
do {
   Leer numeros
   Calcular MCD
   Preguntar si continuar
} while (respuesta afirmativa);
```

Para leer los números haremos a su vez un ciclo de repetición **do while**, mientras que para calcular el máximo común divisor usaremos un ciclo **while**. Por último preguntamos al usuario si quiere continuar y léemos su respuesta. El ciclo para si se lee un carácter 'n'. La razón de que se lee dos veces un carácter es que la primera toma el caracter *return*(nueva-linea) que se introduce después de los dos enteros. A continuación se presenta el código que resuelve el programa.

```
int main(){
   int a,b;
   do {
      do {  //Lee los enteros positivos
         printf("\nCalculo el MCD.\nDime dos enteros positivos: ");
         scanf("%d%d", &a, &b);
      }while(a <= 0 || b <= 0);
      int x = a, y = b;  // Calcula el máximo común divisor
      while(x != y) if (x>y) x = x-y; else y = y - x;
      printf("\nEl MCD entre %d y %d es %d\n", a, b, x);
      printf("\nContinuar? s o n? ");
      getchar();
   } while(getchar() != 'n');
   return 0;
}
```

•

1.13.3. El Ciclo for

El ciclo **for** es la última de las instruciones de repetición del lenguaje C y talvez la más usada. Se usa, por lo general, cuando se conoce de antemano el número de veces que se ha de repetir una secuencia de instrucciones. Su sintaxis es como segue:

```
for (<expr1>;<expr2>;<expr3>) Instruccion
```

donde <expr1>es una expresión que tiene como finalidad inicializar las variables de control del ciclo, <expr2>establecer la condición de parada del ciclo y <expr3>actualizar las variables de control del ciclo. Su interpretación operacional es: "se inicializan las variables indicadas en <expr1>, se evalua la expresión <expr2>y si resulta cierta se ejecuta la instrucción **Instruccion**, finalmente se actualizan las variable de control en base a la expresión <expr3>y se repite el proceso desde la evaluación de la expresión <expr2>. Si al evaluar la <expr2>la misma resulta falsa, se termina la ejecución."

Las expresiones <expr1>y <expr3>son por lo general asignaciones o invocación de funciones, mientras <expr2>es una expresión booleana. Recuerde que en C, cualquier valor no nulo se interpreta como TRUE y el valor CERO se interpreta como FALSE. Cuaquiera de las tres expresiones se puede omitir, pero

no los *punto-y-coma*. Si se omite la expresión <expr2>, la condición de parada se interpreta como TRUE y el ciclo se convierte en un ciclo infinito, a menos que se interrumpa por algún otro medio como un **break** o un **return**.

Salvo por el comportamiento del **continue** que describiremos luego, el ciclo **for** es equivalente al siguiente código:

```
<expr1>;
while (<expr2>) {
   Instruccion
   <expr3>;
}
```

Ejemplo 1.21 (Factorial—de nuevo) *Escribir un programa que permita calcular el factorial de un número natural n.*

Explicación: Debemos declarar la variable n como constante para almacenar el número al que queremos determinar el factorial, una variable k para almacenar el factor actual y una variable r para almacenar los productos parciales. Podemos efectuar el producto empezando por el factor 1 hasta el n o al revés, aquí lo haremos de 1 a n. Usaremos la instrución **for**. Inicializamos la variable r en 1, no necesitamos inicializar la variable k pues lo haremos en el ciclo en CERO. El ciclo se ejecuta mientras k sea menor estricto que n y la variable se incrementa de uno en uno.

```
int main(){
   const int n;
   int k, r = 1;
   for (k=0; k<n; k++) r = r*(k+1);
   printf("\n%d! = %d.", n, r);
   return 0;
}
```

Nótese que si $n = 0$, no se entra al ciclo, y que cada vez que cada vez que se intenta entrar al ciclo r almacena el factorial de k. •

1.14. Comandos break y continue

A continuación presentaremos dos comandos que permiten terminar parcial o totalmente la ejecución de un ciclo. El primero de ellos interrumpe la ejecución del ciclo en el cual se encuentra, el segundo interrumpe sólo la iteración actual del ciclo.

1.14.1. break

A veces conviene interrumpir la ejecución de un ciclo en un lugar que no sea su principio o su fin. El comando **break** sirve para interrumpir la ejecución del

ciclo más interno en el cual se encuentra, de manera similar a como interrumpe la ejecución de un **switch**.

1.14.2. continue

1.15. Ejercicios Resueltos

1.16. Ejercicios

1. Se desea pagar una cierta cantidad entera no mayor de $500, en billetes de las siguientes denominaciones: 20, 10, 5, 2 y 1 de tal manera que se use la menor cantidad de billetes. Escriba un programa en C que resuelva el problema.

2. Dados tres puntos del plano de coordenadas (x_1, y_1), (x_2, y_2) y (x_3, y_3), escriba un programa que permita decidir si forman un triángulo rectangulo.

3. Dados tres enteros a, b, c, escriba un programa en C que permita hallar la mediana de los tres números.

4. Dados dos enteros A, B, con $B \geq 0$, escriba un programa que permita calcular A^B.

5. Escriba un programa en C que permita resolver la ecuación

6. Dado un número entero n, escriba un programa en C que permita invertir el número. Usar sólo aritmetica entera. Por ejemplo, el inverso de 2880 es 882, esto es, no cuentan los ceros a la derecha.

7. La función $\operatorname{sen} x$ se puede evaluar usando su desarrollo en series de Taylor en torno a cero. Este es, $\operatorname{sen}(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$. Por supuesto que no vamos a estar haciendo esta suma toda la vida. La condición de parada puede ser cuando el valor absoluto del nuevo sumando sea poco significativo, por ejemplo, menor que *error* y darle a error un valor pequeño como 10^{-6}.

8. Escriba un programa en C que permita contar el número de divisores positivos del entero positivo n. Haga dos versiones: una que use la instrucción **while** y otra que use la instrucción **for**.

9. Según el teorema fundamental de la aritmética todo número entero no nulo se puede escribir como $\pm 1 \prod primos$. Escriba un programa en C que permita escribir un entero no nulo como el producto de sus primos. Por ejemplo, si el número es 60, la salida debe ser: 60 = +2*2*3*5, si es -8, debe ser: -2*2*2 y si es 1, debe ser +1.

10. Un número natural es palíndrome si se lee lo mismo de derecha a izquieda que de izquierda a derecha. Escribir un programa que permita decidir si un número natural es o no palíndrome.

11. Escriba un programa en C que permita leer de la entrada estandard una por una las notas de un estudiante y dé como salida el promedio, la menor y la mayor de las mismas. Haga dos versiones, en una pida primero el número de notas y en la otra pare cuando se introduzca el valor -1.