PROGRAMACIÓN EN C APLICANDO CONCEPTOS DE FÍSICA

Gabriel Gerónimo C.
Salomón González M.

BORRADOR VERSIÓN 0.2. MAYO 2022.

Universidad Tecnológica de la Mixteca
Instituto de Computación-Física
{gcgero,salomong}@mixteco.utm.mx

Prólogo

Existen muchos libros, y notas en la actualidad que versan sobre el lenguaje C, por lo que no pretendemos competir con ninguna de las aportaciones ya realizadas. Lo que se pretende es iniciar con la familiarización de un lenguaje de programación y su aplicación en los cálculos relacionados con materias básicas de las ciencias exactas que se imparten en las ingenierías.

En la mayoría de las ingenierías se dictan en los primeros semestres materias de programación, y materias de Física. En sus unidades de estudio se discuten los temas básicos de estructuras de control, cinemática y dinámica de Newton respectivamente. Las presentes notas están pensadas para que los estudiantes apliquen el lenguaje C como herramienta de apoyo para comprender y solucionar problemas seleccionados de Cinemática y Dinámica.

A lo largo de las unidades de explicarán primero los temas de programación estructura en lenguaje C, y después se explican temas de Física. Las explicaciones de Física sirvarán como soporte para el desarrollo de propuestas de programación.

Recuerde que este documento es el primer borrador de los primeros capítulos, todavía no es el documento final liberado. Sólo es compartido para nuestros estudiantes de ingeniería en Computación y Física de los primeros semestres de la Universidad Tecnológica de la Mixteca con la finalidad de su lectura para su preparación de sus parciales. Cualquier aportación para enriquecer las notas es bienvenida, por lo que puede escribir a los correos: gcgero@mixteco.utm.mx, o salomong@mixteco.utm.mx

Yo no quiero elogiarte como acostumbran los arrepentidos,
porque te quise a tu hora, en el lugar preciso,
y harto sé lo que fuiste, tan corriente, tan simple,
pero me he puesto a llorar como una niña porque te moriste.
Tia Chofi. Jaime Sabines.

Universidad Tecnológica de la Mixteca
Instituto de Computación, Instituto de Física
M.C. Gabriel Gerónimo C, Dr. Salomón Gozález M.
{gcgero, salomong} @mixteco.utm.mx

CONTENIDO

Capítulo 1

El Lenguaje de Programación C

C es un lenguaje de programación que permite interactuar de forma directa con el usuario, y en forma indirecta con el hardware del dispositivo. En este capítulo se darán las bases para iniciar el arte de la programación, teniendo en consideración que se hará como en la vieja escuela, desde una terminal y aplicando la compilación desde línea de comandos.

Si alguien quiere ser programador, hay un solo camino: elegir software libre y aprender a programar.

Richard M. Stallman.

1.1 Introducción a la programación

El lenguaje de programación C no se encuentra emparejado con ningún hardware ni sistema operativo, por lo que el mismo programa C puede ser ejecutado en cualquier equipo que maneje C. Se hace notar que, al decir “maneje”, significa que el equipo debe contar con un compilador que reconozca el código escrito en C y a partir de él realice la tarea de crear el ejecutable para dicho equipo.

El lenguaje C y su compilador nacieron para el desarrollo e implementación del sistema operativo UNIX[1]. Al evolucionar UNIX y tratar de portar a un sistema operativo similar a él en cualquier hardware nació GNU/Linux, o como lo llaman esencialmente: Linux. Linux es un sistema operativo libre que reconoce, permite y alberga el compilador GCC (GNU Compiler Collection) de C. En el Anexo 1, se coloca un texto del creador del compilador gcc, Richard M. Stallman, fundador del proyecto GNU, donde se puede leer parte de la filosofía de GCC y el software libre. Para realizar y probar los programas de este capítulo y de los capítulos posteriores, se toma por omisión que cuenta con algún equipo que tiene instalado algún sistema operativo UNIX, derivado de UNIX, GNU/Linux o alguna de distribución derivada de GNU/Linux, así como el compilador GCC. Obviamente, aunado a esto, debe contar con un editor de texto para la captura de los códigos de los programas.

El objetivo principal de este curso es aprender cómo programar usando el lenguaje estructurado llamado C y aplicar la teoría de física básica como base para el planteamiento y desarrollo de programas. Para controla el flujo de datos, el lenguaje C proporciona una estructura secuencial, tres de selección y tres de repetición. El modo secuencial o estructura de secuencia, es la más simple de programación, es una sola sentencia o un conjunto de sentencias que se ejecutan una después de otra, y sólo se ejecutan una vez. En el presente capítulo se iniciará el estudio del modo secuencial del flujo de datos, y se dará la introducción de los siguientes seis controles de flujo que se retomarán en el capitulo 2 para aundar más sobre ellos.

Para alcanzar el objetivo planteado se realizará primero una introducción al tema a tratar, se planteará el problema a resolver, se mostrará una propuesta del programa en C para su resolución y se explicarán las instrucciones plasmadas en el código. Cada uno de los programas

1 D. Ritchie (1993). The Development of the C Languaje. Bell Labs/Lucent Technologies. ACM.

se explicará línea por línea, teniendo en consideración que las líneas explicadas con anterioridad no se repetirá su explicación. Lo anterior para poder ir avanzando en los nuevos temas que abarque el capítulo.

Un programa en C está formado por una o más funciones, y a su vez cada función se encuentra formada por una o más proposiciones o sentencias. Cada función se le asigna un nombre, excepto a la función principal llamada ***main***, ella ya tiene asignada su nombre por defecto, así que no se permite cambiarle de nombre. Así también, cada función puede tener parámetros o argumentos de entradas y parámetros o argumentos de salida. Los parámetros de entrada se colocan entre paréntesis después del nombre de la función, y los parámetros de salida se definen antes del nombre y se regresan de la función por medio de la instrucción ***return***. En el capítulo 3 se mostrará con detalle lo que es una función y sus pasos de parámetros.En este capítulo, sólo se explorará la función principal ***main***.

Se debe tener presente que en estas notas se explicará solamente la programación C sobre los sistemas UNIX, LINUX o sistemas derivados. Además de compilar los programas desde la línea de comandos, usando el compilador **gcc**. Para su ejecución también se toma en omisión que se hará desde la línea de comandos. Se debe saber que todo compilador, tiene siempre o casi siempre seis fases para formar un ejecutable, dichas fases se enuncian a continuación:

Fase 1. Editar. Esta primera fase consiste en editar un archivo, ejecutando para ello un programa de edición, que contendrá el código en C conocido como ***código fuente***. Dicho código se almacena en un dispositivo de almacenamiento, y se le debe asignar un nombre y la extensión **.c**. Dentro de los editores de textos básicos, más comunes, que puede ejecutar desde una terminal se encuentran: vi, nano, vim, pico, emac.

Fase 2. *Preprocesar*. La fase de preprocesamiento se ejecuta de forma automática antes de la fase de traducción. Se encarga de realizar comandos especiales llamados directrices de preprocesador, que indican tareas a realizar antes de la compilación. Las tareas, por lo regular, consisten en la inclusión de otros archivos en el archivo a compilar y en el reemplazo de símbolos especiales con texto de programa.

Fase 3. *Compilar*. Se encarga fundamentalmente de realizar las tares de análisis léxico, análisis sintáctico, y análisis semántico del código fuente, además de crear el código objeto.

Fase 4. *Enlazar*. En esta fase, el enlazador se encarga de vincular el código objeto (creado en la fase de compilación) con el código de las funciones faltantes. Lo anterior, con la finalidad de construir una imagen o código ejecutable. Por ejemplo, al escribir en la terminal: gcc hola.c, si la compilación y el enlace se realizan en forma correcta, se producirá un archivo con el nombre de a.out. Dicho archivo será la imagen ejecutable del programa C llamado hola.c.

Fase 5. *Cargar.* Antes de que un programa sea ejecutado, este debe estar colocado en memoria. Esto se realiza por medio del cargador que toma el ejecutable del disco y lo lleva a un espacio de memoria.

Fase 6. *Ejecutar*. Como última fase, la Unidad Central de Procesamiento (CPU) toma la imagen ejecutable cargada en la memoria, y se encarga de realizar el ciclo de ejecución instrucción por instrucción.

Se debe hacer la anotación que todo programa en ejecución se convierte en un proceso que se alberga dentro del sistema. El sistema para identificarlo le asigna un número, y le asocia tres archivos para la comunicación: ***stdin*** (estándar de entrada), ***stdout*** (estándar de salida) y ***stderr*** (estándar de error). El estándar de entrada por lo regular está asignado al teclado, el estándar de salida por lo regular a la pantalla, y el estándar de error, también asociado, por lo regular, a la pantalla.

1.2 Compilación con GCC

Se parte del supuesto que la programación se realiza en un sistema derivado de UNIX. Dentro de ellos, se encuentran GNU/Linux en cualquiera de sus distribuciones y los sistemas de las computadoras Mac. Se recomienda crear un directorio, dentro de su directorio personal de ***home***, donde se alberguen todos los ejemplos de estas notas. La creación del directorio se puede realizar usando un administrador de archivos o desde la terminal usando el comando **mkdir**. Dentro del directorio creado, debe abrir un editor de textos y copiar las líneas de código (excepto la numeración que inicia en cada línea) que forman cada uno de los ejemplos, compilar el programa con gcc (creará el archivo ejecutable), y creado el ejecutable proceder a invocarlo. Suponga que el nombre del programa le llame hola.c, entonces en su línea de comandos deberá escribir:

gcc -Wall hola.c -o hola

Después de que no marque el compilador ningún error, deberá invocar al ejecutable creado, que en este caso fue llamado hola. La invocación será:

./hola

Los caracterés ./ indican al sistema que el ejecutable se encuentra en la ruta actual, y no tiene que ir a buscarlo dentro de las rutas que están predeterminadas en PATH.

Ejemplo 1.
El objetivo del siguiente código es dar a conocer las instrucciones básicas que constituyen a todo programa C. Para llevar una constante en todos los programas básicos desarrollados en estas notas, en las primeras líneas se colocará el nombre propuesto para asignar al archivo, y la forma de compilación a realizar desde la terminal. Se pide transcribir el siguiente código en un archivo de texto, tomando en consideración no transcribir la numeración de cada línea. Esta numeración de líneas servirá después para ubicar la sentencia que se explicará del programa.

```
1  /* ********************************************************************************************
2                Programación en C para escribir un mensaje en pantalla
3                                        hola.c
4                           compilar: gcc -Wall hola hola.c -o hola
5  ******************************************************************************************** */
6  #include <stdio.h>
7
8  int main (int argc, char *argv[])
9  {
10   printf (“Hola, estamos aprendiendo C.\n”);
11   return 0;
12 }
```

Después de capturar el código anterior, guarde el archivo de texto con el nombre propuesto de **hola.c**, dentro del directorio de ejemplos creado en su directorio personal de ***home***. Recuerde que no debe colocar los números de cada línea, estos números servirán en la sección donde se explican las sentencias que forman el cuerpo del programa.

Para poder realizar la compilación, es decir, verificar que el código del programa capturado sea correcto, debe abrir una terminal, ubicarse en el directorio donde está su programa en C, y escribir:

$ **gcc -Wall hola.c -o hola**

Sólo debe escribir lo que está en negrita. El símbolo $ se tomará como el prompt (símbolo antes del cursor) del sistema, omítalo en el momento de capturar los comandos en la terminal. Al escribir lo anterior, se invoca al compilador ***gcc*** indicando que debe revisar el código albergado en el archivo de texto ***hola.c*** y si está libre de errores (por medio de la opción ***-Wall***) deberá crear el

ejecutable o código de máquina. La opción ***-o*** realiza la tarea de crear el código de máquina con el nombre que se coloca a continuación, en este caso, el nombre del archivo ejecutable es **hola**. En caso de omitir la opción ***-o***, la salida es escrita a un archivo, por defecto, llamado ***a.out***. Si existe ya un archivo con ese nombre, solamente reescribirá sobre él. En el caso de no utilizar opciones para el proceso de compilación, el compilador gcc normalmente se hace cargo del preprocesamiento, compilación, ensamblado y enlazado del archivo.
Aunando más sobre la opción ***-Wall***, se puede decir que se encarga de habilitar todos los avisos de error y de peligro que pueden aparecer en la construcción del código. En caso de no aparecer error indica que la compilación fue exitosa, en caso contrario debe abrir el código con su editor y corregir la línea en donde se halla el error.

Suponga entonces que está libre de errores el código del programa, por lo que ya se creo el archivo ejecutable en el directorio actual. Por lo que para verificar que el archivo creado esté en el directorio, escriba en la terminal:

$**ls -l**

ls es un comando del sistema que muestra los archivos que se encuentran en el directorio actual, y si la pasamos el parámetro ***-l***, mostrará detalles de cada uno de los archivos de dicho directorio. Se podrá visualizar entonces que se encuentra el archivo ***hola.c*** y el archivo ejecutable ***hola*** creado por el compilador gcc. Estando seguros que se encuentra el ejecutable, se escribe en la terminal:

$.**/hola**

Lo anterior se encargará de cargar el archivo ejecutable en memoria, y hacer que la CPU inicie el proceso de ejecución de las instrucciones del programa. ***./*** es utilizado para indica al sistema que es el directorio actual donde se encuentra el código a ejecutar. Sino existe error en la invocación del ejecutable, se podrá visualizar en la terminal:

Hola, estamos aprendiendo C.

Si visualiza lo anterior, ¡¡felicidades!!, ha realizado su primer programa en C.

A continuación se analizarán las líneas de código que forman al programa hola.c, por lo que se recurrirá a usar los números asociados a cada línea que se colocó junto con el código.

Líneas 1 a 5:

```
/* ************************************************************************
                Programación en C para escribir un mensaje
                                hola.c
                        compilar: gcc -Wall hola.c -o hola
************************************************************************ */
```

Los caracteres /* y */ son utilizados en C para escribir un conjunto de líneas de comentarios en el código. Por lo que, las líneas que inician con /* y finalizan con */ indican al compilador que es un comentario y no debe tomarse en consideración para el proceso de compilación. Los comentarios se colocan como notas para documentar el código del programa. Cuando se colocan varias líneas de comentarios se debe iniciar con ***/**** y se debe cerrar con ****/*** , en el caso de que el comentario sea una sola línea, basta con colocar al inicio los caracteres de diagonal ***//*** y a continuación escribir el comentario. Los comentarios permiten a otras personas leer y comprender el código del programa, pero recuerde no hacer demasiados comentarios, podría complicar la lectura del código.

Línea 6:

```
#include <stdio.h>
```

Está línea de código le indica al compilador que librería del sistema es necesaria para que sean referenciada en la compilación. En este caso, se referencia a la librería ***stdio.h*** debido a que el programa invoca a la función ***printf***, la cual se encuentra definida dentro de dicha librería. Debe notar que todas las librerías que se necesiten incluir para el funcionamiento del programa, deben iniciar con el símbolo #. Dicho símbolo le indica al preprocesador que es un directiva, y los símbolos **< >** que encierran a **stdio.h** indican que debe buscarla en el directorio donde se encuentran todas las librerías del sistema, por lo general, las librerías están ubicadas en el directorio ***/usr/include***. También puede, en lugar de los símbolos **< >**, colocar los caracteres de doble comillas **“ “** y dentro de ellos el nombre de la librería, y con esto le indicará al preprocesador que dicha librería se debe buscar en el directorio actual. En una sección de este capítulo se explicarán más directivas que puede utilizar en C, y en el anexo 4 puede encontrar información de las librerías usadas con mayor frecuencia en la programación en C.

Línea 8:

```
int main (int argc, char *argv[ ])
```

Está línea es la parte esencial de todo programa en C, indica el inicio de la función principal llamada **main,** con dos argumentos de entrada. El primer argumento de tipo entero llamado argc, es decir, **int argc**, que se encarga de contar el número de argumentos que se pasan a la función principal, y que son albergados en una lista direccionada por el segundo argumento llamado argv, es decir, **char *argv[]**. Los argumentos que recibe la función ***main*** son adquiridos desde la línea de comandos del sistema cuando se invoca al archivo ejecutable. Todas las funciones, incluyendo la principal (siempre llamada **main**) tienen que indicar el tipo de parámetro que retorna o de salida, y el tipo o tipos de parámetros que recibe o de entrada.

Línea 9:

```
{
```

Línea 12:

```
}
```

Todo conjunto de instrucciones o bloques de sentencias deberá ir encerrado entre los símbolos de llave izquierda { y llave derecha }. En este caso, es un bloque que forma a una función. Se debe tener una llave izquierda para indicar el inicio (línea 9) de la función y una llave derecha para indicar el cierre (línea 12) de la función. Dentro de estás llaves se debe colocar las instrucciones que conforman todo el cuerpo del bloque.

Línea 10:

```
printf (“Hola, estamos aprendiendo C.\n”);
```

La línea 10 contiene el llamado a la función ***printf***, la cual es una función de salida de datos, cuyo encabezado se encuentra definida en la librería stdio.h. Está función se utiliza para mandar a imprimir en la pantalla la cadena que se encuentra entre comillas, excepto el carácter de escape que se encuentra como el símbolo de diagonal invertida ****, y la letra **n** (**\n)**, juntos indican a la función **printf** que debe hacer un salto de línea y colocar el cursor al inicio de ella. En el Anexo 2, puede encontrar mayor información sobre las secuencias de escape. Otra de las cosas que se debe notar en está línea es que para finalizar se debe colocar punto y coma, es decir, el carácter **;**. La función ***printf***, es una función que puede mandar la salida de un fragmento de memoria a la impresora, disco, otro programa, o en su defecto a la pantalla, como en este caso.

Línea 11:

```
return 0;
```

La instrucción **return** se utiliza para indicar que el programa terminó de forma satisfactoria y devuelve el valor de 0 al proceso que invocó el ejecutable, en este caso el retorno es al sistema operativo. En caso de no terminar de forma correcta, el ejecutable puede devolver otro valor, o el mismo proceso que lo invocó puede devolver un valor diferente de cero. Note que el retorno es un

tipo de valor entero debido a que la función principal debe devolver ese tipo de parámetro de salida, es decir, como lo indica la línea 8: **int** main(int argc, char *argv[])**.**

Otra forma de indicar que el programa termina en forma correcta o incorrecta, es retornar la constante EXIT_SUCCESS o EXIT_FAILURE. Esta manera elegante de indicarlo, es usando la librería ***stdlib.h***, y las constantes definidas en ella:

```
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0
```

Ejemplo 2.

El objetivo del siguiente código[2] es mostrar la importancia el parámetro -**Wall** en la compilación del código fuente. El parámetro -**Wall** se encarga de habilitar todos los errores de peligros que se puedan presentar en los constructores que forman parte del programa. Al igual que en las indicaciones del programa 1 se le pide transcribir el siguiente código en un archivo de texto, no transcribiendo la numeración de cada línea.

```
1 /* ************************************************************************************************************
2                     Programa C para mostrar Error de peligro
3                                           mal.c
4                      compilar:   gcc -Wall mal.c -o mal
5  ************************************************************************************************************ */
6 #include <stdio.h>
7
8  int main (void)
9  {
10    printf (" 2 + 2 = %f\n", 4);
11   return 0;
12  }
```

Después de capturar el código, realice la compilación:

$**gcc -Wall mal.c -o mal**

Después de escribir lo anterior, podrá ver en pantalla los siguientes mensajes:

```
mal.c: In function ‘main’:
mal.c:10: warning: format ‘%f’ expects type ‘double’, but argument 2 has type ‘int’
```

A pesar del aviso de peligro, note que se genera el ejecutable, para ello, ejecute el comando ***ls*** y verá que generó el compilador el archivo llamado **mal**. Por lo que al proceder a la ejecución del programa llamado mal, es decir, escribir en su terminal:

$**./mal**

Podrá notar la salida del programa en pantalla, y visualizará:

2 + 2 = 0.000000

Lo cual es algo incorrecto, el error de salida es debido a que a la función **printf** se le pasa un número entero, 4, en lugar de un número real, es decir, coma flotante. Las variables de tipo entero y coma flotante son almacenados en diferentes formatos en memoria, por lo que al recuperarlos se obtiene algo incorrecto. Además note la importancia de utilizar en la compilación la opción -**Wall**, la cual se encarga de detectar errores habituales que se tienen al iniciar a programar en C, los cuales, en ocasiones se pasan por alto.

---

2 Para mayor información sobre este código u opciones del compilación en gcc consulte: An Introduction to GCC de Brian Gough.

La manera de corregir el error es escribir la línea 10 como:

```
printf (" 2 + 2 = %f\n", 4.0);
```

Teniendo la pauta anterior, se tocará a continuación los temas de variables, tipos de datos y expresiones.

1.3 Variables

Una variable en lenguaje C, debe recibir un nombre, y para ello se debe iniciar con una letra o un símbolo de subrayado (_) seguida de letras o dígitos. Por lo general deberán ser de una longitud de 31 caracteres. Se debe notar que para el compilador es diferente una palabra con minúsculas que una con mayúsculas, a ello se le llama sensibilidad del compilador. Otra de las consideraciones, es que no se pueden utilizar palabras reservadas del lenguaje como identificadores de las variables, es decir, no debe usar: if, else, do, while, for, int, float, etc., como nombre de una variable. A su vez, el nombre de la variable tiene asociado un tipo de dato. La variable puede almacenar un número (entero, o real), un carácter o una estructura. Antes de utilizar una variable en todo código de C esta debe estar declarada. En forma general la forma de declarar una variable es:

***tipo* identificador1 [= valor], identificador2 [= valor], . . . ;**

En donde:
tipo, puede ser entero, es decir, utilizar las palabras reservadas: int, char, short, long int, unsigned int, unsigned long int; o puede ser real, es decir, utilizar las palabras reservadas: float, double, long double.
identificador1, puede ser el nombre de una variable o varias de ellas separadas por comas.
[valor], si se necesita inicializar la variable con un valor, se le puede asignar dicho valor desde su declaración. Los corchetes ***[ ]*** solo son usados para indicar que se puede omitir lo que se encuentra entre ellos.

Por ejemplo:

```
int a, b=10;
```

La anterior línea, indica que es la declaración de una variable de tipo **int (entero)** llamada **a**, sin inicializar, y la declaración de una variable de tipo **int (entero)** llamada **b** inicializada con el valor de **10**.

```
float x,r,z;
```

La línea anterior, indica la declaración de tres variables de tipo **float**, llamadas **x**, **r**, **z**, respectivamente.

Un programa en C, cualquiera que sea su tamaño, contiene por lo menos un bloque de sentencias, y ésta contiene variables para almacenar información. Las sentencias especifican las operaciones de cálculo que se van a realizar, y las variables almacenan los valores utilizados durante el cálculo. Cuando se realizan los cálculos, las sentencias deben terminar con punto y coma (;). Aunque se pueden colocar más de una sentencia en una línea y estas deben estar separadas por el punto y coma. Se recomienda escribir una sola sentencia por línea para hacer más claro el código en el momento de una lectura.

De acuerdo con su ámbito las variables pueden ser variables locales o variables globales, es decir, conocidas en un ámbito local o en un ámbito global. Para ello se utilizan los modificadores de clases de almacenamiento: automático (***auto***), externo (***extern***), estático (***static***), y de registro (***register***).

1.3.1 Variables locales

Todas las variables declaradas dentro de una función se le llaman variables locales, y sólo son conocidas cuando se ejecuta dicha función. Las variables reservan espacio de memoria que sólo es utilizado en el transcurso de su ejecución, al finalizar dicha ejecución, la memoria es liberada y las variables son destruidas, por la razón anterior, a estas variables se les llama variables automáticas (***auto***). La especificación ***auto*** por lo general no es utilizada para declarar variables locales debido a que por defecto las variables locales tienen dicha cualidad. Las variables pasadas como parámetros de entrada en las funciones son también variables locales, excepto si se le pasa una dirección de memoria, lo cual hace que la misma variable apunte a ese espacio y se pueda modificar los datos ahí almacenados, en ese caso, sólo las variables se destruyen pero no es liberada la memoria asociada a ella.

Existe también la declaración ***static*** para las variables dentro de una función, la característica que tiene esta declaración es que conservará su valor cuando se termine la ejecución de la función, y cuando se invoque nuevamente dicha función se podrá recuperar el valor de dicha variable.

1.3.2 Variables globales

Las variables definidas fuera de las funciones, por lo general definidas después del área de las inclusiones, son conocidas por todas las funciones que forman parte del programa. Estas variables se mantienen y pueden modificarse en cada acceso a ella, además sus espacios de memoria asignados son destruidos al finalizar el programa. Una buena práctica para su uso en las funciones es declararlas como externas, es decir, usar la palabra reservada ***extern*** para indicar que es un variable global, previamente declarada, que será usada en la función. Por ejemplo:

```
....
int global;

int main (void)
{
 extern int global;
...
}
```

Indica que aunque la variable entera llamada global ya fue previamente declarada, la función principal main hará uso de ella en su bloque y podrá modificarla.

Ejemplo 3.
El objetivo del siguiente código es introducir la librería ***math.h*** y el parámetro de compilación **-lm**. La librería **math.h** es utilizada en C para incluir funciones matemáticas predefinidas y el uso del parámetro ***-llibrería*** indica al compilador que en la fase de enlazado se debe incluir la **librería** indicada.

```
1 /* ********************************************************************************************************
2                         Programa uso de la librería de matemáticas math.h
3                                 raiz.c
4                         compilar:  gcc -Wall raiz.c -lm -o raiz
5 ******************************************************************************************************** */
6 #include <stdio.h>
7 #include <math.h>
8
9 int main (void)
10 {
11  double x=2.0,y;
12
```

```
13  y=sqrt (x);
14 printf (" Raiz cuadrada de %lf = %lf\n", x,y);
15 return 0;
16 }
```

Después de guardar el archivo, se debe invocar al compilador con las opciones ya vistas anteriormente, pero ahora incluyendo la opción **-lm**

$ **gcc -Wall hola.c -lm -o hola**

La opción **-lm** indica que se deberá anexar para la compilación la librería de matemáticas. En forma general se escribe **-l*librería***, donde librería puede ser: ***m*** (**-lm**) para incluir el enlace de la librería de matemáticas, ***termcap*** (**-ltermcap**) para incluir librería de manejo de terminales, ***curses*** (**-lcurses**) para incluir librería de manejo de ventanas, ***l*** (**-ll**) para incluir librería para el uso de lex[3] y yacc[4].

Línea 7: #include <math.h>
Indica al preprocesador que incluya las funciones matemáticas que se encuentran dentro del archivo **math.h**. Siempre que se incluya esta librería, al compilar se le debe indicar usando la opción **-lm**.

Línea 11: double x=2.0,y;
En esta línea se hace una declaración de dos variables de tipo doble coma flotante. Como se observa, a la variable **x** se le asigna un valor (**x=2.0**) desde su declaración. Esto puede facilitar la inicialización y el ahorro de líneas de código.

Línea 13: y=sqrt (x);
En esta línea se invoca a la función **sqrt**(), la cual se encuentra definida en ***math.h***, y cuya sintaxis es la siguiente:

PROTOTIPO DE LA FUNCIÓN

```
#include <math.h>

double sqrt(double x);
```

La función **sqrt** recibe como parámetro de entrada una variable de tipo double, y retorna un valor del mismo tipo. En el anexo 3 puede consultar un conjunto de funciones básicas de matemáticas que se encuentran declaradas en la librería **math.h**.

Línea 14: printf (" Raiz cuadrada de %lf = %lf\n", x,y);
En esta línea se manda a imprimir a pantalla el valor de **x** y el resultado del cálculo de la raíz cuadrada almacenado en **y**. Note que para el tipo de dato double se debe colocar %lf.

1.4 Tipos de datos

C soporta tres tipos básicos de datos: enteros, coma flotantes (o reales) y caracteres. De ellos se desprenden, los tipos de datos con signo, sin signo, cortos, largos y dobles. En la tabla 1 se definen los tipos de datos utilizados en C.

3 Librería para apoyar en la programación de un analizador léxico.
4 Librería para apoyar en la programación de un analizador sintáctico (parser)

Tabla 1. Tipos de datos.

Tipo de dato	Palabra reservada	Formato usado con printf	Formato usado con scanf
Carácter	char	%c	%c
Entero corto	short	%hd	%hd
Entero	int	%d	%d
Entero sin signo	unsigned int	%u	%u
Entero largo	long int	%ld	%ld
Entero largo sin signo	unsigned long int	%lu	%lu
Real (coma flotante)	float	%f	%f
Real doble	double	%lf	%lf
Real largo	long double	%Lf	%Lf

El lenguaje C no define tamaño fijo para sus tipos de datos básicos, lo único que se garantiza es: short int ≤ int ≤ long int; unsigned ≥ 0; y float < double < long double.

Se debe tener presente que un valor de un tipo superior puede ser convertido a un inferior de dos maneras: 1) asignando el valor a una variable de tipo inferior, o 2) mediante el uso del operador ***cast***[5]. Aunando un poco más en **cast**, el operador **cast** es un operador unario que es asociado de derecha a izquierda con la misma precedencia que otros operadores unarios. Se forma colocando el nombre del tipo de datos entre paréntesis antes de la variable que se le aplicará el **cast**.

Ejemplo 4.
El objetivo del siguiente programa es reconocer el tamaño de los tipos de datos que tiene el sistema, así como la identificación de sus rangos. Es recomendable saber el tamaño del almacenamiento de cada tipo de dato que permite la arquitectura del equipo de cómputo en la cual se ejecutarán sus programas, esto permitirá evitar el desbordamiento de las variables (buffer overflow). Dicho desbordamiento puede suceder en el momento de realizar ciertos cálculos, y se pueden presentar en la etapa de ejecución del programa dando como resultado un error muy peligroso que puede dañar la vulnerabilidad de los programas y del sistema.

```
1  /* ********************************************************************************************************
2                 Programa uso de sizeof() y constantes para verificar
3                    tamaño y rangos de tipos de datos del sistema
4                     compilar: gcc -Wall tamano.c -o tamano
5  ******************************************************************************************************** */
6  #include <stdio.h>
7  #include <limits.h>
8  #include <float.h>
9
10 int main (int argc, char *argv[])
11 {
12   printf ("char=%ld bytes\t unsiged=%ld\n", sizeof (char), sizeof(unsigned));
13   printf ("short int=%ld bytes\t int=%ld bytes\n",sizeof (short int), sizeof (int));
14   printf ("float=%ld bytes\t long=%ld bytes\t",sizeof(float),sizeof(long));
15   printf ("double=%ld bytes\t long double=%ld\n", sizeof(double), sizeof(long double));
```

5 cast o casting es una conversión explícita de un tipo de dato a otro. P.e. int a; printf (“%f”,(float)a);

```
16
17    printf ("Rango de entero: %d\t%d\n", INT_MIN, INT_MAX);
18    printf ("Rango de Long: %ld\t%ld\n", LONG_MIN, LONG_MAX);
19    printf ("Rango de float:%f\t%f\n", FLT_MIN, FLT_MAX);
20    printf ("Rango de double:%E\t%E\n", DBL_MIN, DBL_MAX);
21    printf ("Rango de double largo:%Lf\t%Lf\n", LDBL_MIN, LDBL_MAX);
22    return 0;
23 }
```

Línea 12-15:
En estas líneas se hace uso de la función ***sizeof()*** para mandar a imprimir el tamaño en bytes de los tipos de datos definidos en C. La función recibe como parámetro el nombre de una variable o el nombre del tipo de dato, y se encarga de retornar su tamaño en bytes.

Línea 17-21:
En estas líneas se mandan a imprimir a pantalla las constantes que albergan los valores simbólicos mínimo y máximo asignados a los tipos de datos que utiliza el sistema. Las constantes: INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, se encuentran definidas en la librería **limits.h**, mientras que las constantes: FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX, LDBL_MIN, LDBL_MAX, se encuentran definidas en la librería **float.h**.

En la línea 20 se debe notar que para mandar a imprimir a pantalla un número en notación científica (o potencia de 10) se utiliza %E.

1.5 Expresiones

Una expresión consiste en un conjunto de elementos identificados como operandos y operadores. Estas expresiones pueden ser aritméticas, lógicas y de carácter. El resultado de una expresión aritmética es un valor numérico, el resultado de una expresión lógica es un valor booleando, 0 para indicar falso ó 1 para indicar verdadero, y el resultado de una expresión de carácter es un carácter o conjunto de caracteres. En la tabla 2 se muestran los operadores aritméticos simples y compuestos aplicados en C, y en la tabla 3 los operadores lógicos. En C existe un operador llamado operador de asignación, cuyo símbolo es =, que se encarga de colocar un valor en una variable. Internamente lo que se realiza es colocar dicho valor dentro de una dirección de memoria asignada por el administrador de memoria del sistema, y asociar a la variable con esta dirección.

Tabla 2. Operadores aritméticos simples y compuestos.

Operador	Símbolo	Ejemplo	Retorna
Operadores Simples			
Suma	+	b=a+10	El resultado de la suma del valor de **a** más 10 en la variable **b**.
Resta	-	b=a-10	El resultado de la resta del valor de **a** menos 10 en la variable **b**.
Multiplicación	*	c=a*b	El resultado de la multiplicación del valor de **a** por b en la variable **c**.
División	/	c=c/a	El resultado de la división de **c** entre **a** en la variable **c**.
Módulo	%	a=a%2	El residuo de la división de **a** entre 2 en la variable **a**.

Operador	Símbolo	Ejemplo	Retorna
Operador de pre incremento	++	++a	El valor de **a** incrementado en 1 en la misma variable **a**.
Operador de post incremento		a++	El valor actual de **a** y después incrementa y lo coloca en la misma variable **a**.
Operador de pre decremento	--	--a	El valor de **a** decrementado en 1 en la misma variable **a**.
Operador de post decremento		a--	El valor actual de **a** y después decrementa en 1 y lo coloca en la misma variable **a**.
Operadores Compuestos			
Si b y c son variables entonces la expresión b = b * c será equivalente a b *= c	*=	b*=c	El resultado de la multiplicación de **b** por **c** en la variable **b**.
Si b y c son variables entonces la expresión b = b / c será equivalente a b /= c	/=	b/=c	El resultado de la división de **b** entre **c** en la variable **b**.
Si b y c son variables entonces la expresión b = b % c será equivalente a b %= c	%=	b%=2	El residuo de la división de **b** entre **2** en la variable **b**.
Si b y c son variables entonces la expresión b = b + c será equivalente a b += c	+=	b+=c	El resultado de la suma del valor de **b** más **c** en la variable **b**.
Si b y c son variables entonces la expresión b = b - c será equivalente a b - = c	-=	b-=c	El resultado de la resta del valor de **b** menos **c** en la variable **b**.

Tabla 3. Operadores lógicos.

Operador	Símbolo	Ejemplo	Retorna
Operador booleano de comparación			
Igual que	==	a==10	Un valor de 1 que indica verdadero en caso de la variable **a** sea igual a **10** y 0 en caso falso.
Diferente que	!=	a!=10	Un valor de 1 que indica verdadero en caso de la variable **a** sea diferente a **10** y 0 en otro caso.
Mayor o igual que	>=	a>=10	Un valor de 1 si la variable **a** es mayor o igual a **10** y 0 en otro caso.
Menor o igual que	<=	a<=10	Un valor de 1 si la variable **a** es menor o igual a **10** y 0 en otro caso.
Mayor que	>	a>10	Un valor de 1 si la variable **a** es mayor a **10** y 0 en otro caso.
Menor que	<	a<10	Un valor de 1 si la variable **a** es menor a **10** y 0 en otro caso.
AND	&&	a && b	Un valor de 1 si **a** y **b** no son cero y 0 en otro caso. Esta operación se evalúan de izquierda a derecha.
OR	||	a || b	Un valor de 1 si al menos un elemento de la variable **a** o de la variable **a** no son cero y 0 en otro caso. Esta operación se evalúan de izquierda a derecha.
NOT	!	!a	Un valor de 1 si el valor de **a** es 0, y 0 en todos los demás casos.
Operador booleano a nivel bit			
AND	&	a = a & b	El resultado del AND entre **a** y **b** en la variable **a**.
XOR	^	a = a ^ b	El resultado del XOR entre **a** y **b** en la variable **a**.
OR	|	a= a | b	El resultado del OR entre **a** y **b** en la variable **a**.
Complemento a uno	~	a = ~ a	El resultado del complemento a uno de **a** en la variable **a**.
Desplazamiento a la izquierda	<<	a= a << i	El resultado del desplazamiento de **a** a la izquierda **i** posiciones en la variable **a**.
Desplazamiento a la derecha	>>	a = a >> i	El resultado del desplazamiento de **a** a la derecha **i** posiciones en la variable **a**.

1.5.1 Precedencia de operadores

La precedencia de operadores es utilizada por C para realizar los cálculos de las expresiones en un orden correcto. Casi todos los operadores se asocian de izquierda a derecha, a excepción del

operador de asignación que se asocia de derecha a izquierda. En la tabla 4 se muestran los operadores utilizados en C, y su asociatividad aplicada al realizar las evaluaciones de las expresiones.

Tabla 4. Operador de precedencia.

Operador ordenados por precedencia	Asociatividad
( )	Se calculan primero. Se realiza el cálculo de izquierda a derecha.
*, /, %	Se evalúan los cálculos en segundo lugar. Si existen varios se realizan los cálculos de izquierda a derecha.
+, -	Se realiza el cálculo al final. Si existen varias evaluaciones se realizan de izquierda a derecha.
<, <=, >, >=	Se realizan de izquierda a derecha.
==, =!	Se realizan de izquierda a derecha.
=	Se realizan de derecha a izquierda.

1.6 Instrucciones básicas de entrada y salida

Toda entrada y salida de los programas es realizada por medio de un flujo de caracteres y es terminada por medio de un carácter de nueva línea o fin de archivo (EOF). Como se mencionó en una sección anterior, cuando un programa es ejecutado el sistema operativo le asocia tres flujos de datos para comunicarse con él: ***stdin***, ***stdout*** y ***stderr***. El sistema operativo tiene la característica de permitir que estos flujos sean redirigidos a otros programas, por medio de conductos (llamados también tuberías o pipes). Por ejemplo al escribir en la terminal:

$ ***programa*** < ***archivo***

Se está indicando al sistema operativo, por medio del símbolo **<**, que ***programa*** tome como entrada el contenido de ***archivo***. Por otra parte, al escribir en la terminal:

$ ***archivo*** **| programa**

Se está indicando al sistema, por medio del símbolo **|**, que la salida de ***archivo*** será la entrada de ***programa***.

Los dos ejemplos anteriores son mecanismo de interconexión que brinda el sistema para redirigir la entrada y salida de datos. Si solamente se invocado al programa ejecutable, se toma por omisión que la entrada de datos es el teclado, y la salida de datos es la pantalla.

1.6.1 Función: printf

La función **printf**, definida en la librería **stdio.h**, es utilizada para controlar el formato de salida de datos. Por medio de ella se puede especificar conversión (%), banderas, tamaño de campos, precisión, tipo de dato (vea tabla 1) y carácter de escape (vea Anexo 2), a todo este conjunto se le conoce como *control de formato*. La sintaxis de la función **printf** indica que el control de formato debe colocarse entre comillados (“ “), y después se deberá escribir la lista de argumentos necesarios para obtener información requerida por los campos de control de formato. La función retorna un número entero que indica el número de caracteres que fue impreso. En forma general su sintaxis es:

```
PROTOTIPO DE LA FUNCIÓN
    #include <stdio.h>

    int printf(const char *controldeformato, ...);
```

El campo control de formato puede verse de la siguiente manera, donde lo que se coloca entre [ ] indica que puede colocarse o puede omitirse:
% [***banderas***][***tamaño de campos***][***.precisión***] **tipo de dato** [***carácter de escape***]

Donde los campos anteriores pueden contener lo siguiente:
banderas

- Un signo menos (-) para indican que se debe realizar un ajuste de la salida a la izquierda.
- Un signo mas (+) para indicar que se debe imprimir el signo + cuando es número es positivo.
- El número 0 para indicar que se antepongan ceros para llenar el tamaño del campo. Cuando se usa 0 se debe también usar el campo **tamaño de campos**.
- tamaño de campos
- Un número entero que se encarga de reservar el tamaño de lugares de salida para escribir el número que imprimirá la función **printf**.

.precisión

- Un punto (.) seguido de un número entero, para indicar a **printf** que debe escribir el número de decimales especificado por el valor del entero. Por ejemplo .2, indica que debe escribir dos decimales.

tipo de dato.

- Especifica el tipo de argumento que se imprimirá (vea tabla 1). Por ejemplo: %f, indica que se imprimirá un tipo de dato real, y este valor lo tomará de lista de argumentos que se coloca después de cerrar las doble comillas.

carácter de escape.

- Especifica que se mandará a imprimir un carácter de escape (vea anexo 2).

1.6.2 Función: scanf

La función **scanf**, definida en la librería **stdio.h**, permite adquirir información de la entrada estándar, la cual normalmente es el teclado. Dicha entrada es interpretada según las especificaciones que estén en cadena de control y son almacenadas en las variables que se colocan en su lista de argumentos. En forma general su sintaxis es:

```
PROTOTIPO DE LA FUNCIÓN
    #include <stdio.h>

    int scanf (const char *controldeformato,listadeargumentos);
```

En forma general la función tiene dos parámetros de control para la entrada. El primero de ellos es el ***control de formato*** de entrada, en la cual se debe colocar entre comillados (“ “) el o los tipos de datos (vea tabla 2) que se esperan recibir en la entrada, y el segundo que es la ***lista de argumentos***, que se coloca fuera de las comillas, es o son los nombres de las variables que deben iniciar con ampersand, donde se almacenaran los tipos de datos (vea tabla 1) leídos. El ampersand tiene la función de dirigir el o los valores leídos a la dirección de memoria asignada a la o las variables, por ese motivo al ampersand se le conoce como operador de dirección. Excepto cuando se mandará a imprimir a pantalla una variable de tipo cadena, debido a que ella tiene implicita la dirección donde está albergado los caracteres.

1.6.3 Funciones: **getchar** y **putchar**

Las funciones **getchar** y **putchar**, se encuentran especificadas en la librería **stdio.h**. **getchar** permiten leer carácter por carácter de la entrada definida. Su sintaxis es la siguiente:

```
PROTOTIPO DE LA FUNCIÓN
    #include <stdio.h>

    int getchar(void);
```

La función es invoca sin argumentos, y retorna el carácter leído en formato entero o EOF cuando encuentra fin de archivo. EOF es una constante que se encuentra definida en **stdio.h**, cuyo valor es -1.

Otras funciones que puede utilizar para salida de caracteres y cadenas son: fgetc, fgets, getc, ungetc, cuyas sintaxis son:

```
PROTOTIPO DE LA FUNCIÓN
    #include <stdio.h>

    int fgetc(FILE *stream);
    char *fgets(char *s, int size, FILE *stream);
    int getc(FILE *stream);
    int ungetc(int c, FILE *stream);
```

Por otra parte, **putchar**, se encarga de imprimir en la salida predeterminada el carácter que se le pasa como parámetro. **putchar** retorna el carácter escrito o EOF si ocurre algún error. Su sintaxis es la siguiente:

```
PROTOTIPO DE LA FUNCIÓN
    #include <stdio.h>

    int putchar(int c);
```

Otras funciones que puede utilizar para salida de caracteres o cadena son: fgetc, fgets, getc, ungetc, cuyas sintaxis son:

```
PROTOTIPO DE LA FUNCIÓN
    #include <stdio.h>

    int fputc(int c, FILE *stream);
    int fputs(const char *s, FILE *stream);
    int putc(int c, FILE *stream);
    int puts(const char *s);
```

En las anteriores funciones, las variables de tipo ***FILE***, indican que son variables de tipo archivo, y dependiendo de la función sirven como entrada o salida de caracteres del archivo.

1.7. Inclusiones

En esta sección se mostrarán directivas del preprocesador que se utilizan en C para indicar que se deben incluir contenido de otros archivos, códigos de macros, segmentos de código o eliminación de segmentos de códigos, durante el proceso de la compilación del programa. Las

directivas indicadas al preprocesador inician con el símbolo # y tradicionalmente son escritas al inicio el código de los programas.

1.7.1 #include

La directiva **#include** es usada para incluir códigos de librerías u otros programas en el programa donde se incluye dicha directiva. La sintaxis puede ser:

#include <*archivo*.h>

La sintaxis anterior indica que el preprocesador debe buscar el ***archivo.h*** cuyo nombre se colocado entre los símbolos **< >** en la ruta definida por el sistema donde se incluyen todas las librerías relacionadas con el lenguaje. Típicamente en los sistemas UNIX la ruta es ***/usr/include***. Por convención los nombres de las librerías tienen la extensión **.h**

Otra sintaxis puede ser:

#include "*archivo.h*"

Esta directiva indica al preprocesador que busque en el directorio actual el ***archivo.h*** cuyo nombre se colocado entre comillas " ", en caso de no encontrarlo el compilador imprimirá un mensaje de error y terminará en forma anormal su ejecución.

Lo que realiza el preprocesador en los dos casos anteriores es el reemplazo de **#include directiva** por el contenido del archivo.

1.7.2 #define

Una línea de código que inicia con **#define** se utiliza para indicar la definición de una constante simbólica o un macro. Si se va a definir una constante, la sintaxis es la siguiente:

#define *nombre* *reemplazo*

En la sintaxis ***nombre*** es el identificador asignado a la constante, la cual sigue las reglas comunes para nombrar identificadores en C. Cualquier ocurrencia de ***nombre*** después de la definición es reemplazado por lo que aparezca en ***reemplazo.*** Ejemplos de definiciones pueden ser:

```
#define PI      3.1416
#define ANCHO       20
#define LARGO       ( ANCHO + 40 )
```

A partir de las anteriores definiciones, cualquier ocurrencia en programa donde aparezcan los identificadores PI, ANCHO o LARGO serán reemplazado por su asociación correspondiente.

La sintaxis para un macro es:

#define *nombre(argumento_1, ... ,argumento_n)* reemplazo

Si un macro con argumentos es expandido, cualquier ocurrencia de un argumento es reemplazado por el valor actual de los argumentos que se colocan en **reemplazo**. Los macros permiten realizar tareas simples pero no complejas, si son complejas debe utilizar funciones. Por ejemplo un macro simple:

#define MAY(a,b) (a>b) ? a:b

El macro anterior es un ejemplo de macro con parámetros de entrada, por lo que para invocarlo se deben colocar los argumentos definidos. Suponga que se tiene el siguiente fragmento de código, donde se invoca al macro anterior:

```
int x=10, y=20;
printf ("El mayor es %d\n", MAY(x,y));
```

Cuando se realice el proceso de preprocesamiento, la línea de código donde aparece el macro, **MAY(x,y)** será reemplazado por:
printf ("El mayor es:%d\n", **(x>y) ? x:y**);

También se debe tener en consideración que los nombres de los macros deben estar construidas al igual que los identificadores, con una secuencia de letras y dígitos que inicien con una letra o con un carácter de subrayado (_). Sólo que por convención, es recomendable utilizar como identificador caracteres en mayúsculas, con la finalidad de no confundirlos con variables. Normalmente un macro se define en una sola línea, sin embargo, si un macro es más largo que una línea, se debe utilizar el símbolo **** al final de cada línea para indicar que continua su declaración, excepto la línea final que debe ser continua. Por ejemplo:

```
#define IMPRIMIR (valor, mensaje) \
   if (valor) printf (mensaje)
```

El preprocesador cuenta con macros predefinidos, algunos de ellos son: __LINE__, __FILE__, __DATE__, __TIME__, __STDC__, __VERSION__, entre otros. Note que cada nombre es precedido y sucedido por dos caracteres _.

1.7.3 #if, #else, #endif

Las directivas **#if**, **#else**, y **#endif** se pueden usar para que el preprocesador condicionalmente incluya o excluya partes de código en un programa. La directiva **#if** se encarga de evaluar una expresión constante entera. Si esta constante es verdadera se incluye las sentencias que se encuentra antes de **#endif**. La sintaxis es:

```
#if expresión
   sentencias;
#endif
```

En caso de querer incluir sentencias cuando ***expresión*** sea falsa, se debe hacer uso de **#else**, cuya sintaxis es:

```
#if expresión
        sentencias;
  #else
        sentencias;
#endif
```

Por ejemplo, el siguiente fragmento de código le indica al compilador definir un macro llamado TRUE con el valor de 1, e indicar en el preprocesamiento que cuando TRUE tiene determinado valor asigne un valor a ESTADO.

```
#define TRUE 1
#if TRUE == 1
        ESTADO = 100;
#else
        ESTADO = 200;
#endif
```

1.7.4 #undef

La directiva **#undef** se utiliza para eliminar la definición del nombre de un macro, el cual fue definido previamente. La sintaxis es:

```
#undef NOMBREDELMACRO
```

1.7.5 #ifdef, #ifndef, #endif

Las directivas **#if define** es equivalente a **#ifdef**, y **#if !define** es equivalente a **#ifndef**. La directiva **#ifdef** es utilizada para definir el código de un macro. Cuando se realiza el proceso de compilación del código fuente de un programa, y el compilador es invocar con el parámetro **-DNOMBRE**, este comprueba si un macro con cierto **NOMBRE**, ya fue definido y si es así, se debe incluir en el preprocesamiento las líneas de código que se encuentran antes de la directiva **#endif**. Por ejemplo si se tiene el siguiente fragmento de código en algún programa dentro de la función principal **main**:

```
#ifdef PRUEBA
  printf (“Prueba...\n”);
  printf (“.....\n”);
#endif
```

La compilación del código fuente (donde se incluyo el fragmento anterior), se debe hacer usando de la opción **-DPRUEBA** para indicar que se debe incluir en el preprocesamiento las líneas de código entre **#ifdef** y **#endif**:

```
$ gcc -Wall -DPRUEBA hola.c -o hola
```

Así que, en forma general en la compilación se debe de invocar a la opción _**D***NOMBREMACRO* para definir el macro a incluir en el preprocesamiento llamado *NOMBREMACRO*.

Si el programa es compilado sin **-D** entonces las sentencias después de **#ifdef** son omitidas después del preproceso, y el ejecutable no incluirá el código, en caso contrario se busca y se incluye el código.

En el sistema existe un conjunto de macros predefinidos que se pueden utilizar en el preprocesamiento. Si desea conocerlos, los puede listar invocando al preprocesador con la opción **-dM**, como se muestra a continuación:

```
$ cpp -dM /dev/null
```

Se hace notar que **cpp** es el preprocesador de macros de C usado automáticamente por el compilador gcc para transformar el programa antes de la compilación.

Ejemplo 5.

El objetivo del siguiente programa es utilizar el flujo redirigido del sistema (**<**) para poder alimentar la entrada de la función de lectura **getchar**(). También se muestra la programación básica de un macro, y la forma de compilar cuando se utiliza.

```
1  /* ************************************************************************************************************
2  *              Programa semejante a la salida del
3  *                              comando wc
4  *      compilar: gcc -Wall -DPRUEBA WC.c -o WC
5
************************************************************************************************************ */
6  #include <stdio.h>
7  #define V       1  // Letra de una palabra
8  #define F       0  // No es letra
9
10  int main()
11  {
12        #ifdef PRUEBA
13          printf ("LEYENDO archivo...\n");
```

```
14            printf (".....\a\n");
15       #endif
16   int c,nl,nc,np,estado;
17   estado=V;
18   nl = nc = np = 0;     // contadores a cero
19
20   while ( (c=getchar()) != EOF)
21   {
22     ++nc;
23     if ( c== '\n')
24        ++nl;
25      if ( c== ' ' || c == '\n' || c=='\t')
26       estado=V;
27      else if (estado == V)
28       {
29         estado=F;
30         ++np;
31         }
32     }
33    printf("Palabras=%d, Lineas=%d, Caracteres=%d\n", np, nl+1, nc);
34    return 0;
35  }
```

Después de la captura del código, debe almacenar el archivo con el nombre de WC.c, y proceder a escribir en la terminal:

$ **gcc -Wall -DPRUEBA WC.c -o WC**

La opción **-D** indica al preprocesador que debe incluir el macro que a continuación se le indica, en este caso el macro fue llamado **PRUEBA**.

Para la ejecución del programa se debe alimentar la entrada desde un archivo, por lo que se debe escribir:

$ **./WC < nombre_archivo**

Por ejemplo, si se usa el mismo archivo fuente como entrada para alimentar **getchar**(), puede escribir:

$ **./WC < WC.c**

Por lo que verá como salida del programa en la terminal, lo siguiente:

```
LEYENDO archivo...
.....
Lineas=38, Palabras=86, Caracteres=731
```

Línea 20: while ((c=getchar()) != EOF)
Está línea se encarga de leer carácter por carácter los datos almacenados en el archivo WC.c, y se detiene hasta que encuentra el fin de archivo (EOF). En el siguiente capítulo se verá mayor detalle del ciclo de repetición **while**, y la estructura de decisión **if-else**.

Capítulo 2

Estructuras de control aplicando Cinemática

En el lenguaje C a los componentes principales se llaman sentencias. Las sentencias pueden ser expresiones que terminan con punto y coma, estructuras de control de flujo, o bloques de instrucciones que son encerradas entre llaves { }. El lenguaje cuenta con siete estructuras de control, clasificadas como: una de secuencia, tres de selección (if, if-else, switch), y tres de repetición (while, for, do-while). El modo secuencial o estructura de secuencia, es la más simple de programación, es una sola sentencia o un conjunto de sentencias que se ejecutan una después de otra, y sólo se ejecuta una vez. En este capítulo se estudiarán las estructuras de selección y las estructuras de repetición que brinda el lenguaje C, se definirán conceptos de Cinemática y basados en ello se planearán soluciones de programación para realizar cálculos relacionados con el tema.

Si algo fue sencillo en tu mente, debió ser sencillo cuando lo escribiste en código.

Brian Kernighan

2.1 Estructuras de selección: if, if-else, switch

En C existen tres estructuras para realizar una selección de instrucciones a ejecutar. La instrucción **if** es la más simple, permite realizar un conjunto de sentencias siempre y cuando una expresión sea verdadera. La instrucción **else** es el complemento de **if**, debido a que toma en consideración el caso cuando la expresión lógica resulta falsa. En casos en los cuales se necesite verificar más de dos condiciones, se utiliza la instrucción **switch**.

2.1.1 if

La instrucción **if** se encarga de evaluar una expresión lógica, que se coloca después de **if** entre paréntesis, si dicha evaluación resulta verdadera entonces se realizan la o las sentencias que se encuentran después de la expresión. La sintaxis de la instrucción **if** es:

```
if (expresión) {
        sentencias;
        }
```

Se debe notar que si son más de una las sentencias que se deben ejecutar cuando la expresión evaluada por **if** es verdadera, estas deben colocarse entre llaves { }, en el caso que sea una sólo sentencia a ejecutar se puede o no colocar las llaves, lo decide el programador.

2.1.2 if-else

Cuando se evalúa una expresión esta puede ser verdadera o falsa, en el caso de querer tomar una decisión cuando la expresión resulte falsa, se debe utilizar la instrucción **else**, y en seguida ejecutar las sentencias que se requieran. La sintaxis de **if-else** es:

```
        if (expresión) {
                sentencias;
                }
        else  {
                sentencias;
                }
```

2.1.3 if-else anidados

En algunas ocasiones se necesita realizar evaluaciones de expresiones dentro de las sentencias que forman parte del bloque **if** o del bloque **else**. Cuando sea este el caso, se colocan **if** dentro de los **else**, construyendo formatos de multi bifurcación. La sintaxis de **if-else** anidados se puede escribir como:

```
if (expresión) {
   sentencias;
  }
   else if (expresión) {
          sentencias;
        }
   else if (expresión) {
          sentencias;
        }
else {
  sentencias;
}
```

2.1.4. Operador ternario ?:

El lenguaje C cuenta con un operador ternario que se encarga de sintetizar lo que realiza un **if-else** con una única sentencia de ejecución. El operador utiliza tres operadores, el primero es la expresión lógica a verificar, el segundo es la sentencia a ejecutar cuando la condición es verdadera y el tercero es la sentencia a ejecutar cuando la condición es falsa. En seguida se muestra con pseudocódigo, y a continuación de ella un extracto de código.

El pseudocódigo para **if-else** simple es:
if (expresión) sentencia_1;
else sentencia_2;

Mientras que el pseudocódigo para el operador ternario es:
expresión **?** sentencia_1 **:** sentencia_2;

Un extracto de código usando **if** puede ser:
if (a>b) printf (“ a es mayor”);
else printf (“b es mayor”);

Lo cual se sintetiza en una sola línea con el operador ternario:
a>b **?** printf (“a es mayor”) **:** printf (“b es mayor”);

2.1.5. switch

La intrucción **switch** es una estructura de selección múltiple que es muy útil cuando se requiere seleccionar una de muchas alternativas que puede tomar una variable. El valor que puede adquirir la variable a cuestionar debe ser entero o carácter, debido a que sólo se verifica si es igual a o

pertenece a un conjunto. Para verificar el valor de una variable se escribe la instrucción **switch** y entre paréntesis la variable a cuestionar, después de ello, se escribe entre llaves { } cada uno de los casos que puede tomar la variable, así como también un caso por omisión o default, si es que se necesita tener un caso que se ejecutará cuando la variable no tome ningún valor de los casos anteriormente planteados. La sintaxis de **switch** es:

```
switch (variable)
{
  case etiqueta_1:
                    sentencias;
  case etiqueta_2:
                    sentencias;
   ....
  case etiqueta_n:
                   sentencias;
  default:                          // opcional
          sentencias;
 }
```

Cada etiqueta contiene el valor que puede tomar la variable colocada después de la palabra reservada **switch**. Dicha etiqueta puede ser una letra, que debe colocarse entre comillas simples, o un valor entero que no debe colocarse entre comillas, dicho valor es comparado contra lo que contiene la variable, cuando alguna de ellas resulte verdadera se ejecutarán las sentencias que se encuentren dentro del **case**, en caso que no coincida ninguna de las etiquetas, se ejecutarán las sentencias que se encuentran en **default**. Se hace notar que si se coloca **break**, lo que hace es que no realice las comparaciones de todos los casos, es decir, si uno de los casos es verdadero, entra y ejecuta las sentencias, y al encontrar **break** termina el **switch** y continua con la sentencia que se encuentra fuera de la estructura. La sintaxis de **switch-break** es:

```
switch (variable)
{
  case etiqueta_1:
                    sentencias;
                    break;          // se ejecuta y termina el switch
 case etiqueta_2:
                    sentencias;
                    break;          // se ejecuta y termina el switch
  ....
  case etiqueta_n:
                   sentencias;
                   break;
  default:                          // opcional
          Sentencias;
 }
```

2.2 Estructura de repetición: while, do-while, for

El lenguaje C soporta diferentes estructuras de repetición, que permiten que un conjunto de sentencias se ejecuten repetidamente hasta que una cierta condición se cumpla. Las estructuras de repetición pueden estar controladas por una expresión que se evalúa en cada ciclo, a estas estructuras se les conoce como ciclo centinela; también se encuentran las estructuras donde se conoce con anticipación el número de veces que se realizará el ciclo, a estas estructuras se le conoce como ciclo contador. Las estructuras de repetición con que cuenta C son: **while**, **do**-**while** y **for**, a continuación se explicará cada una de ellas.

2.2.1 while

La instrucción **while** forma una estructura de repetición que se encarga de evaluar en cada ciclo si una expresión lógica es 1 (verdadera), si es así se encargará de ejecutar de cero a n veces las sentencias que se colocan a continuación dentro del bloque. Al igual que en i**f**, si son más de una sentencia se deben colocar entre llaves { } para que se estas sean ejecutadas. Cuando la expresión llega a ser 0 (falsa), ya no se ejecuta el bloque y continua con la primera sentencia que se encuentra fuera del **while**.

La sintaxis del **while** es:

```
while ( expresión ) {
        Sentencia_1;
        Sentencia_2;
        ...
        Sentencia_n;
  }
```

2.2.2 do-while

La característica de esta estructura de repetición es que por lo menos una vez son ejecutadas las sentencias que se encuentran dentro de su bloque, esto debido a que la expresión a ser evaluada se coloca al final. Después de la evaluación de la expresión, si esta es 1 (verdadera), regresa a ejecutar la primera sentencia dentro del bloque, en caso de que la expresión sea 0 (falsa), continuará con la sentencia que se encuentre fuera del bloque.

La sintaxis del **do-while** es:

```
do
{
  Sentencia_1;
 Sentencia_2;
  ...
 Sentencia_n;
} while (expresión);
```

2.2.3 for

La instrucción **for** es un constructor de C para crear un ciclo de repetición de sentencias, combina la *inicialización* de una variable, una evaluación de una *expresión lógica,* relacionada con la variable inicializada, y una *actualización* de la variable previamente inicializada. La *inicialización* de la variable se realiza antes de iniciar el ciclo (sólo se inicializa una vez), la *expresión lógica* es evaluada en cada repetición del ciclo, si es 1 (verdadera) se ejecutan las sentencias, después de ello, se realiza la *actualización* de la variable y se retorna al inicio del control de ciclo para una nueva evaluación lógica. Cuando la expresión lógica evaluada llega a ser 0 (falsa) se deja de realizar las sentencias que se encuentran dentro del bloque, y se ejecuta la siguiente sentencia que está fuera del ciclo formado por **for**. La sintaxis de **for** es:

```
for (inicialización; expresión; actualización)
{
        Sentencia_1;
        Sentencia_2;
        ...
        Sentencia_n;
}
```

2.3. Cinemática -Descripción del movimiento

En la programación que se mostrará en esta sección se tomarán en consideración aspectos de Física, por lo que se inicia estudiando los temas básicos de cinemática y a continuación se plasma un código donde se apliquen las fórmulas y teoría vista. El objetivo que se persigue es, aplicar los conceptos de física junto con las estructuras de control de flujo mostrados en las secciones anteriores.

La cinemática es el estudio de la descripción del movimiento de los objetos sin atender las causas que lo producen. Ejemplos típicos son la caída libre de objetos bajo gravedad, tiro parabólico, desplazamiento de un objeto con aceleración constante, entre otros muchos. Los objetos al moverse cambian su posición a lo largo del tiempo, a esto se le conoce como velocidad.

La velocidad promedio se define como el cambio en la posición o desplazamiento con respecto al intervalo de tiempo empleado en el recorrido de dicho desplazamiento. Matemáticamente esta velocidad promedio es expresada como:

$$\bar{v} = \frac{desplazamiento}{tiempo\ empleado} = \frac{\Delta x}{\Delta t}$$

En donde $\Delta x = x_2 - x_1$ representa el cambio en la posición del objeto, cuya posición final es x_2 y su posición inicial es x_1 , y $\Delta t = t_2 - t_1$ es el intervalo de tiempo empleado en recorrer ese desplazamiento. La posición x_2 ocurre al tiempo t_2 y la posición x_1 al tiempo t_1. Se usa el símbolo “ $\bar{v}$ ” con barra para distinguir esta velocidad de la velocidad instantánea v. Si se plasma la velocidad en una gráfica posición contra tiempo, la variable de posición es representada en el eje vertical (eje x) y la variable independiente llamada tiempo en el eje horizontal (eje y).
A continuación se muestra un código para realizar el cálculo de la velocidad promedio, aplicando la fórmula propuesta anteriormente, y aplicando la estructura de selección.

Ejemplo 6.

El objetivo del siguiente programa es aplicar las formulas de velocidad promedio, y decidir si el objeto va hacia adelante o hacia atrás.

```
1 /* ************************************************************************************************************
2                     Programa Velocidad Promedio
3                               v_promedio.c
4       compiladar:  gcc -Wall v_promedio.c -o vp
5 ************************************************************************************************************ */
6 #include <stdio.h>
7 #include <stdlib.h>
8
9   int main (void)
10 {
11   float X2,X1;                  // Variables para el desplazamiento en X
12   float T2,T1,Ax,At,Vp;         // Variables para el tiempo, recuerde que T2 > T1
13
14   system ("clear");
15   printf ("Destino X2="); scanf ("%f",&X2);
16   printf ("Origen X1="); scanf ("%f",&X1);
 17  printf ("Tiempo T2="); scanf ("%f",&T2);
 18  printf ("Tiempo T1="); scanf ("%f",&T1);
```

```
 19  Ax= X2-X1;              // Delta_X
 20  At= T2-T1;              // Delta_T
 21  Vp = Ax/At;             // Velocidad_promedio = Delta_X / Delta_T = dx/dt
 22  printf ("Velocidad Promedio=%f m/s\n", Vp);
 23  if (Vp>0) printf ("El objeto se mueve hacia adelante\n");
 24  else printf ("El objeto se mueve hacia atrás\n");
 25  return 0;
26  }
```

Línea 14: *system ("clear");*
La función **system** se encarga de enviar el argumento que se le pasa como parámetro de entrada a la línea de comandos para que sea interpretado por el shell del sistema. En este caso, se pasa el parámetro clear, que es un comando del shell para limpiar la pantalla de la terminal. La función system se encuentra definida en la librería stdlib.h por lo que es indispensable incluirla en el código.

Línea 23:
Para verificar si la velocidad promedio es positiva, se utiliza la instrucción de selección if, y se manda a imprimir el mensaje indicando el movimiento del objeto. En el movimiento unidimensional, la velocidad es positiva, si el objeto se mueve en la dirección en que x crece o negativa, si se mueve en la dirección opuesta. Se recuerda que la magnitud es siempre positiva.

Línea 24*:*
En el caso de que la velocidad promedio sea negativa, se utiliza la instrucción **else**, que es complemento de **if**, y se imprime el mensaje de que el objeto se mueve hacia atrás.

2.4. Caída libre

La caída libre se presenta cuando un objeto se deja caer libremente (sin influencia de algún otro cuerpo) a la superficie de la tierra, y sólo influye sobre el objeto la fuerza de gravedad. Esta fuerza de gravedad tiene una aceleración (aceleración = Fuerza / Masa) dirigida siempre hacia el centro de la Tierra. En nuestro planeta su valor es de a=g=9.8 m/s^2.

Se debe tener presente las siguientes leyes:

1. Todo cuerpo que cae libremente tiene una trayectoria vertical.
2. La caída de los cuerpos es un movimiento uniformemente acelerado.
3. Todos los cuerpos caen con la misma aceleración.

En un sistema coordenado cartesiano, la aceleración se orienta a lo largo del eje y, dirigida siempre verticalmente hacia abajo. Las ecuaciones de la cinemática son:

$$v = v_0 + g \cdot t$$

$$\Delta y = v_0 \cdot t + \frac{1}{2} g \cdot t^2$$

En donde v representa la velocidad final del objeto al tiempo t , v_0 es la velocidad de la partícula en el momento en que inicia su movimiento (tal tiempo inicial es definido de valor cero en las ecuaciones anteriores). El cambio en la posición en la caída del objeto es representado por $\Delta y = y - y_0$ donde y(t) es la posición final al tiempo t y y_0 es la posición inicial al tiempo t=0.

Algunos casos de interés son:

a) El objeto que se deja caer desde el reposo desde una determinada altura. Es conveniente elegir como sentido positivo el movimiento hacia abajo sobre el eje vertical. En este caso es conveniente elegir y_0=0, desde donde es soltada el objeto. Así, la variable y representa la distancia recorrida de

caída libre. Como estamos asumiendo que se suelta desde el reposo, entonces $v_0=0$ Las ecuaciones que rigen la velocidad y cambio en posición del objeto son:

$$v = g \cdot t$$

$$\Delta y = \frac{1}{2} g \cdot t^2$$

Uno puede determinar la velocidad v del objeto para cualquier tiempo posterior a la caída desde el reposo. Así mismo, uno puede determinar la posición de la caída para ese tiempo de caída.

b) Si es lanzada con cierta velocidad inicial v_0 distinta de cero, se debe recurrir al primer conjunto de ecuaciones. Uno debe establecer tal velocidad inicial, por ejemplo, $v_0 = 2\,m/s$. Si el objeto es lanzado hacia arriba (porque también puede ser lanzada hacia abajo), se debe tener cuidado con los signos en las ecuaciones anteriores. Si elige el origen de coordenadas como punto de lanzamiento, entonces el desplazamiento es hacia arriba. Si elege el sentido positivo hacia arriba. Con esto, la aceleración es dirigida hacia abajo y por lo tanto será negativa. Las ecuaciones que gobiernan el movimiento del objeto son:

$$v = v_0 - g \cdot t$$

$$\Delta y = v_0 \cdot t - \frac{1}{2} g \cdot t^2$$

La figura 1 describe ambos comportamientos.

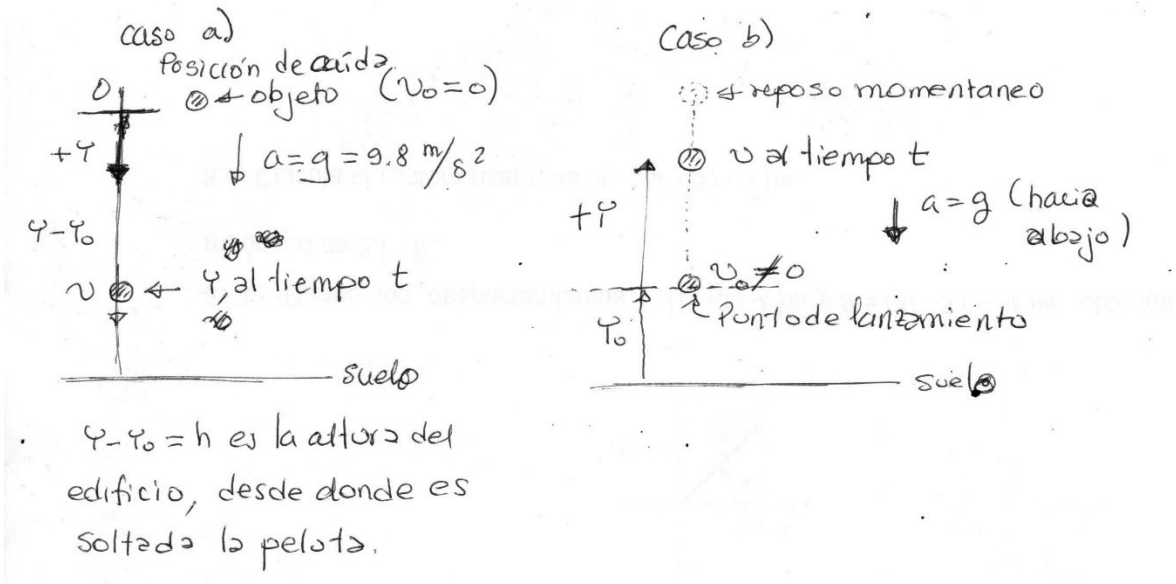

Figura 1. Caída libre de un objeto de masa m . a) se suelta desde el reposo desde cierta altura h=y-yo sobre el suelo. El origen de coordenadas es donde se suelta. Se elige +y hacia abajo. b) El objeto es lanzado hacia arriba con velocidad inicial vo distinta de cero, desde cierta altura yo sobre el suelo. Se elige +y hacia arriba, entonces g apunta hacia abajo y entonces es negativa.

Ejemplo 7.

Para los dos casos presentados, estas ecuaciones pueden ser introducidas en el lenguaje de programación C, para determinar numéricamente los valores de velocidad y posición, en cualquier tiempo dado, sujetas a las condiciones iniciales. Por ejemplo, para el caso b), puede ser lanzado desde la posición inicial y_o=a medida desde el suelo, cualquier valor en metros. Aquí, el programador establece la velocidad inicial de lanzamiento. Una vez establecido el programa y las

condiciones iniciales, se puede conocer la posición final, así como su velocidad de subida para cierto tiempo, antes de llegar momentáneamente al reposo, para luego caer libremente bajo gravedad, desde ese punto de reposo momentáneo. En este punto, el movimiento es descrito como en a).

```
1 /* ************************************************************************************************************
2                         Program Caida libre
3                                 caida.c
4                         gcc -Wall caida.c -o caida
5 ************************************************************************************************************ */
6 #include <stdio.h>
7 #include <stdlib.h>
8 #define G 9.8   // Definimos gravedad de la tierra 9.8 m/s²
9
10 int main ()
11 {
12  float Vf, Vo=0, t=0; // Partimos del reposo
13  system ("clear");
14  printf ("\tTiempo(t)\t Velocidad Final (VF)\n");
15 for (t=0; t<=10; t++)
16  {
17      Vf = Vo + G*t;
18       printf ("\t%f\t\t%f\n",t,Vf);
19   }
20   return 0;
21 }
```

2.5 Ley de gravitación

La ley de gravitación universal de Newton puede ser enunciada como sigue: “Toda partícula en el Universo atrae a otra partícula con una fuerza que es directamente proporcional al producto de sus masas e inversamente proporcional al cuadrado de la distancia entre ellas, y está dirigida a lo largo de la línea que las une”

Si las partículas tienen masas m_1 y m_2 están separadas por una distancia r, la magnitud de la fuerza gravitacional es:

$$F_G = -G\frac{m_1 \cdot m_2}{r^2}$$

Donde G es una constante Universal (el signo menos indica que es dirigida hacia el centro de la Tierra) que recibe el nombre de Constante Gravitacional Universal, la cual al medirse experimentalmente tiene un valor en el Sistema Internacional (SI) de:

$$G = 6.672 \times 10^{-11}\frac{Nm^2}{kg^2}$$

El valor de la gravedad en la superficie de un planeta depende de la masa y radio del planeta. Entonces la aceleración de un cuerpo que es:

$$a = \frac{F}{m} = G\frac{M}{R^2}$$

Lo cual resulta ser la aceleración de la gravedad, de esta manera si se conoce la masa y el radio de cada planeta del sistema solar, se puede encontrar la aceleración en cada uno de ellos (vea Tabla 5).

Tabla 5. Aceleración de la gravedad en cada planeta del sistema solar.

Planeta	Masa (kg)	Radio (m)	Aceleración (m/s^2)
Mercurio	$3.3x10^{23}$	$2.4x10^{6}$	3.8
Venus	$4.9x10^{24}$	$6.1x10^{6}$	8.9
Tierra	$6.0x10^{24}$	$6.4x10^{6}$	9.8
Marte	$6.4x10^{23}$	$3.4x10^{6}$	3.7
Júpiter	$1.9x10^{27}$	$7.1x10^{7}$	23.1
Saturno	$5.7x10^{26}$	$6.0x10^{7}$	9.0
Urano	$8.7x10^{25}$	$2.6x10^{7}$	9.0
Neptuno	$1.0x10^{26}$	$2.5x10^{7}$	11.0
Plutón	$1.3x10^{22}$	$1.2x10^{6}$	0.6

Ejemplo 8. El objetivo del siguiente código es aplicar la teoría anteriormente expuesta. Se solicita elegir un planeta de la lista, después su masa y su radio, y se calcula su aceleración.

```
1   /* ************************************************************************************************************
2                               Programa aceleración gravitacional de planeta
3                                           acelera.c
4                               compilar:  gcc -Wall acelera.c -o acelera
5   ************************************************************************************************************ */
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <math.h>
9   #define G_C 6.672
10   #define G_E -11
11   int main ()
12  {
13   float C_M,C_R,a_c;
14   int E_M,E_R,a_e;
15   char planeta;
16   system ("clear");
17   printf ("Que planeta?:");scanf("%c",&planeta);
18   switch(planeta)
19  {
20   case 'm':
21        printf ("Mercurio\n");
22        break;
23   case 'v':
24        printf ("Venus\n");
25        break;
```

```
26   case 't':
27        printf ("Tierra\n");
28        break;
29   case 'a':
30        printf ("Marte\n");
31        break;
32   case 'j':
33        printf ("Jupiter\n");
34        break;
35   case 's':
36        printf ("Saturno\n");
37        break;
38   case 'u':
39        printf ("Urano\n");
40       break;
41   case 'n':
42        printf ("Neptuno\n");
43        break;
44   case 'p':
45        printf ("Pluton\n");
46       break;
47   default:
48        printf ("No programado\n");
49        return EXIT_FAILURE;
50  }
51 printf ("Lectura de Masa en kg del planeta debe asignar los datos en notacion exponencial\n”);
52  printf ("Coefienciente y Exponente de potencia base 10\n");
53  printf ("Coefiente="); scanf("%f",&C_M);
54  printf ("Exponente base 10="); scanf("%d",&E_M);
55 printf ("Lectura del Radio(m) del planeta debe asignar los datos en notacion exponencial\n");
56  printf ("Coefienciente y Exponente de potencia base 10\n");
57  printf ("Coefiente="); scanf("%f",&C_R);
58  printf ("Exponente base 10="); scanf("%d",&E_R);
59  a_c=(G_C*C_M)/(pow(C_R,2));
60  a_e=(G_E+E_M)-(E_R*2);
61  printf ("Masa (kg)=%.1fx10^%d\n",C_M,E_M);
62  printf ("Radio (m)=%.1fx10^%d\n",C_R,E_R);
63  printf ("aceleracion (m/s^2)=%.2fx10^%d\n",a_c,a_e);
64  return EXIT_SUCCESS;
65 }
```

Capítulo 3

Funciones

Las funciones son un recurso que tiene el lenguaje C para poder hacer modular los programas, permitiendo al programador reutilizar códigos para ahorrar tiempo en el desarrollo de software. En este capítulo se verán las características que brinda el lenguaje para crear funciones, así como los tipos de pasos de parámetros que pueden ser usados para comunicarse con ellas.

El lenguaje C es peculiar en muchos sentidos, pero él, como muchas cosas con éxito, tiene una cierta unidad de enfoque que se deriva de su desarrollo en un pequeño grupo.

Dennis M. Ritchi

3.1 Funciones

Las funciones son la pieza principal de todo programa en C. Como se ha visto en los capítulos anteriores, lo que nunca debe faltar en la estructura de un programa es la función principal llamada *main*, que como se indicó en el capítulo 1 siempre lleva ese nombre. A partir de ella, se pueden invocar a más funciones, que pueden estar en una librería del sistema, librería programada por el programador o contenida en el mismo programa. Al dividir un programa en bloques de funciones, lo que se está realizando es una programación modular. Cada bloque debe ser identificado por lo que se necesita que tenga asignado un nombre, y las sentencias que lo forman deben realizar un conjunto de acciones especificas. La idea detrás de ello es poder hacer reutilizable los códigos de las funciones, es decir, poder utilizarlos en los desarrollos de los demás programas.

En los capítulos anteriores se han usado funciones estándares definidas dentro de librerías de apoyo para el rápido desarrollo de programas en C. Ejemplo de ello, es el uso de la función *printf* que como se ha visto devuelve un entero después de su ejecución (a esto se le llama parámetro de salida), y se le indica entre paréntesis lo que va a mandar a imprimir a pantalla (a esto se le llama parámetro de entrada). De la misma manera, el programador puede diseñar sus propias funciones, para ello debe hacer uso de la siguiente sintaxis:

<***ParametrodeSalida***/***void***> Nombre_Función (<***ParametrodeEntrada***/***void***>)
{
  Sentencias;
  return <expresión>;
}

Donde ***ParametrodeSalida*** es el tipo de dato que puede retornar la función. El retorno de un valor por parte de la función se realiza por medio de la instrucción ***return***. En la sintaxis anterior se coloca la palabra ***expresión*** después de la instrucción return, esta debe coincidir con el tipo de dato declarado en ***ParametrodeSalida***. Si la función no retorna valor, se debe colocar la palabra reservada ***void***, tanto en ***ParametrodeSalida*** como en ***expresión***.

Nombre_Función es el nombre que se le asigna a la función, y esta debe iniciar con una letra seguida por un conjunto de letras, dígitos o subrayado, que servirá para identificar a la función para ser invocada en cualquier parte del programa donde se necesite.

ParametrodeEntrada es el conjunto de parámetros con sus respectivos tipos de datos, que son pasados a la función para que pueda realizar ciertas acciones, si son más de un parámetro cada uno debe ir separado por una coma. Existen funciones que no necesitan parámetros de entrada por lo que este espacio entre paréntesis se coloca vacío o se coloca la palabra reservada ***void***.

Las funciones son declaradas, primeramente, por medio de sus prototipos, los cuales se colocan después de la inclusión del conjunto de librerías a utilizar en el programa. Un prototipo es el encabezado de la función, sin el cuerpo de este, ni los nombres de los parámetros de entrada, sólo sus tipos. La sintaxis es:

ParametrodeSalida *Nombre* (***Tipos de Entrada***)**;**

El declarar el prototipo es fundamental para que el compilador verifique el acceso a la función de forma adecuada con lo que respecta a sus parámetros de entrada y de salida.

Ejemplo 9. El objetivo del siguiente código es utilizar la teoría de la segunda ley de Newton y crear una función para encontrar la fuerza con los datos de masa y aceleración. Para ello, se introduce la información aplicar.

La segunda ley de Newton indica que la aceleración de un objeto en dirección de una fuerza resultante es directamente proporcional a la magnitud de la fuerza e inversamente proporcional a la masa, es decir:

$$aceleración = Fuerza / masa$$

$$Fuerza = masa * aceleración$$

```
1  /* ********************************************************************************************************
2                  Programa para calcular la Fuerza
3                          fuerza.c
4                  compilar: gcc -Wall fuerza.c -o fuerza
5  ******************************************************************************************************** */
6   #include <stdio.h>
7   #include <stdlib.h>
8   float Fuerza (float, float); // Prototipo de la función
9   int main ()
10  {
11   float m,a;
12   system ("clear");
13   printf ("Valor de masa:"); scanf ("%f",&m);
14   printf ("Valor de aceleracion:"); scanf ("%f",&a);
15   printf ("Fuerza=%f\n",Fuerza(m,a));
16  return EXIT_SUCCESS;
17  }
18
19  float Fuerza (float masa, float aceleracion)
20  {
21   return (masa*aceleracion);
22  }
```

3.2. Paso de parámetros en las funciones

Pasar parámetros a una función significa colocar los tipos de datos entre paréntesis después del nombre de dicha función. Para este propósito se tienen dos formas de hacerlo: 1) Paso de parámetros por valor, y 2) Paso de parámetros por referencias. Una explicación rápida para distinguir cada uno de ellos es para el primero sólo se pasa una copia de los valores originales, y para el segundo se pasa la dirección donde se albergan los datos.

3.2.1 Paso de parámetros por valor

Al pasar los parámetros por valor lo que se realiza es una copia del valor del argumento y las modificaciones a la copia no afectan al valor original de la variable. Se debe recordar que las variables declaradas en cada función son variables locales, es decir, solo son conocidas en dicha función. De esta manera cuando invocamos a una función y se le pasa como argumentos las variables, estas sólo copian su contenido en las variables declaradas dentro de la función. El nombre de las variables cuando es invocada la función y en dentro de la función pueden ser los mismos, pero eso no quiere decir que se afecten a las dos. Sólo se afectan a las declaradas dentro de la función. Este es el motivo por el cual se le llama paso de parámetros por valor, sólo se le pasa el valor de la variable. En el siguiente fragmento de código se pasan las copias de las variables ***m*** y ***a*** a la función llamada Fuerza, y está recibe los valores y los almacena en las variables ***x*** y ***y***.

```
int main ()
{
  float F,m,a;

 ....
 m=100.2; a=10.0
 F=Fuerza (m,a);
 ….
}

float Fuerza (float x, float y)
{
  return (x*y);
 }
```

Cuando retorna la función **Fuerza**, ***m*** y ***a*** conservan el mismo valor de cuando fueron invocada la función, es decir, ***m*** contendrá 100 y ***a*** contendrá 10, respectivamente.

3.2.2 Paso de parámetros por referencia

Al pasar los parámetros por referencia se pasa la dirección de la memoria donde se encuentra ubicado el contenido de la variable por lo que al modificar la variable se modifica en realidad el contenido de la memoria.

```
int main ()
{
  float F,m,a;

 ....
 m=100.2; a=10.0
 F=Fuerza (&m,&a);
 ---
}
```

```
float Fuerza (float *x, float *y)
{
 return (x*y);
 }
```

3.3. Función recursiva

Se le llama función recursiva a la función que se invoca a sí misma. Una característica fundamental para las funciones recursivas es que todas deben tener un caso base que es el encargado de indicar de manera explícita cuando la función deja de invocarse a sí mismo. El caso base es el caso mas simple de todo problema que quiera plantearse en forma recursiva, y el programador debe reconocerlo y programarlo por medio de una estructura de selección simple.

Toda función al momento de ser invocada reserva para todas sus variables espacio de memoria para almacenar la información de cada una de ellas, cuando se llama a si misma, crea una nueva reservación de memoria, por lo que se debe tener cuidado de cuantas llamadas se realizarán para no abarcar tanta memoria para una sola función recursiva. Cuando se alcanza el caso base, el último llamado de la función es terminado y liberada su memoria, y hace su retorno a la función que invocó, y así consecutivamente hasta alcanzar el primer llamado de la función.

```
/*************************************************************************************************************
Explicar por medio de un problema de física, como se construye una función recursiva. La propuesta es hacer la programación sin recursión y compararla con recursión. También se debe explicar cual sería el caso base.
*************************************************************************************************************/
#include <stdio.h>
int binario(int);
int main()
{
	int num;
	scanf ("%d",&num);
	printf ("%d",binario(num));
}

int binario(int n)
{
	if (n==1)
	  return (n%2);
	else {
		printf ("%d",binario(n/2));
		return (n%2);
	}
}
```

3.4 Divide y venceras-Compilando archivos fuentes

Un programa en C puede ser dividido en múltiples archivos fuentes, tanto para facilitar su compresión como para su compilación.

Capítulo 4

Arreglos

Las funciones son un recurso que tiene el lenguaje C para poder hacer modular los programas, permitiendo al programador reutilizar códigos para ahorrar tiempo en el desarrollo de software. En este capítulo se verán las características que brinda el lenguaje para crear funciones, así como los tipos de pasos de parámetros que pueden ser usados para comunicarse entre ellas.

Si la depuración es el proceso de eliminar errores, entonces la programación debe ser el proceso de introducirlos.

E. Dijkstra.

4.1 Introducción

En algunas ocasiones se necesitan tener agrupados datos del mismo tipo y refereciarlos con una sola variable, y así no declarar cada una variable por separado, para ello, el lenguaje C permite la declaración de una sola variable con un espacio de memoria reservado para los elementos de su conjunto, a ello le llama *arreglo*. La sintaxis de su escritura es la siguiente: escribir el tipo de dato seguido del nombre de la variable y entre corchetes cuadrados [ ] un número entero que indica la cantidad de elementos que contiene el arreglo, por ejemplo:

```
int arreglo[10];
```

La declaración anterior indica que se tiene una variable llamada ***arreglo*** de tipo entero que albergará a 10 elementos. Se debe hacer notar que el primer elemento del arreglo es el que se encuentra en la posición 0, por lo que el último elemento del arreglo será el ubicado en la posición 9. Así tenemos en forma generar que si un arreglo tiene n elementos, el primero de ellos es el 0 y el último de ellos es el que se encuentra en la posición n-1.

Ejemplo 10.
El objetivo del programa es mostrar el uso de un arreglo. Se inicializa con 0 cada una de sus localidades, después se le colocan multiplos de 2 y al final se muestran en pantalla.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define TAMANO 10
3   int main()
4   {
5     int i,j=2;
6     int arreglo[TAMANO]={0};
7     for (i=0;i<TAMANO;i++)
8      arreglo[i]=i*j;
9     for (i=0;i<TAMANO;i++)
```

```
10    printf(“%d * %d =%d”,j,i,arreglo[i]);
11   return EXIT_SUCCESS;
12  }
```

Línea 6:
Para declarar un arreglo primero se coloca el tipo de dato que albergará la variable arreglo en cada una de sus posiciones. En este caso el tipo es entero, la variable se llama arreglo y es de tamaño TAMANO que es una constante de valor 10. Los arreglos pueden ser inicializados desde su declaración al igual que una variable, pero si se desea inicializar todas las localidades del arreglo con un solo valor, este valor se coloca entre llaves {}, en este caso se inicializa de manera explicita todo el arreglo con 0, es decir, en la posición arreglo[0]=0, hasta arreglo[9]=0.

Línea 8:
Para colocar datos en una localidad del arreglo se debe especificar la posición donde se almacenará dicho dato, esto se realiza por medio del nombre del arreglo, seguido de [y dentro el número entero que indicará la posición, y después cierre del corchete]. En la línea 8 se hace uso del índice i que se inicializa en cero y se va incrementando con la iteración hasta llegar a TAMANO-1, y en dichas localidades almacenará 0, 2, 4, 6,…, 18 respectivamente.

Nombre del arreglo y posición	Contenido en cada posición
arreglo[0]	0
arreglo[1]	2
arreglo[2]	4
arreglo[3]	6
arreglo[4]	8
arreglo[5]	10
arreglo[6]	12
arreglo[7]	14
arreglo[8]	16
arreglo[9]	18

Línea 10:
En está línea se manda a imprimir los valores de las variables j e i, y también el valor contenido en cada posición de la variable arreglo, note que para ello se escribir el tipo de dato que se almacena (%d) y su respectiva variable y posición que se coloca entre corchetes cuadrados (arreglo[i]). Los corchetes cuadrados son considerados en C como un operador, y tienen el mismo nivel de precedencia que los paréntesis.

4.2 Paso de arreglos a funciones

Para pasar como parámetro de entrada un arreglo a una función, se coloca el nombre del arreglo sin corchete, como cualquier variable cuando se pasa a una función. Por ejemplo suponga que se tiene una función que se llama ***Modificar***, y se le ve pasar la variable ***arreglo*** de tipo entero de tamaño 10. Cuando se invoca a la función solo se debe escribir:

Modificar (***arreglo***);

Cuando coloquemos su línea de prototipado se debe escribir:

void ***Modificar*** (int [ ]);

Se debe notar que los corchetes se colocan vacios, y que el lenguaje C simula que el paso de los arreglos es por referencia. Por lo cual, cuando en la función se modifique algún elemento del

arreglo esto modificará al arreglo real en sus localidades de memoria asignados. Por otro lado, si solo pasa un elemento del arreglo este es pasado solo por valor.

Ejemplo 11.
El objetivo del programa es mostrar el paso de un arreglo a una función.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define TAMANO 10
4   void Modificar(int []);
5   int main()
6   {
7      int i=0,j=2;
8      int arreglo[TAMANO]={0};
9       printf("Dirección del arreglo:%p",arreglo);
10     printf ("Dirección de arreglo[%d]:%p",i, arreglo[0]);
11       for (i=0;i<TAMANO;i++)
12        arreglo[i]=i*j;
13     Modificar (arreglo);
14    for (;i<TAMANO;i++)
15     printf("%d * %d =%d",j,i,arreglo[i]);
16   return EXIT_SUCCESS;
17  }
18     void Modificar(int b[])
19     {
20        int i, j=3;
21        printf ("Dirección de arreglo b:%p",b);
22        for (i=0;i<TAMANO;i++)
23          b[i]=i*j;
24        return;
25       }
```

Línea 9, Línea 10:
Note que cuando se manda a imprimir la dirección de memoria del arreglo esta es la misma que la dirección de su posición [0]. Desde ahí inicia la memoria reservada para dicho arreglo.

Línea 21:
Note que cuando se manda a imprimir la dirección de memoria del arreglo b, es la misma que la dirección del arreglo pasado como parámetro, indicando que los dos estan en la misma dirección como si fuese paso por referencia.

Línea 23:
Se está afectando a cada uno de los valores almacenados en la posición i del arreglo, por lo que en la línea 15 se imprimirán dichas afectaciones debido a lo indicado de que un arreglo es una forma de paso por referencia.

4.3 Ordenamientousando arreglos

Una de las actividades fundamentales en la programación es el ordenamiento de los datos, para ello existen algoritmos para realizar dicha actividad. La eficiencia del algoritmo es medida con respecto al tiempo que tarda en resolver el ordenamiento de los datos. Para el ordenamiento se debe tener un conjunto de datos y aplicar determinado orden, ascendente o descendente, con respecto a uno de los elementos del conjunto. Dicho conjunto de datos puede estar almacenado en un archivo, en un arreglo, en una lista enlazada o en un árbol. En esta sección se tomará que

los datos están almacenado en un arreglo, por lo que los algoritmos planeados serán escritos en relación a ello.

La eficiencia del algoritmo mide la calidad y rendimiento del algoritmo, para dicha eficiencia se toman en consideración dos aspecto, menor tiempo de ejecución o menor número de instrucciones. Lo que se busca entonces por lo general, es el menor tiempo posible para realizar la ordenación, para ello se cuenta el número de veces que se realiza una operación de comparación entre los elementos del arreglo. Por lo que se dirá que un algoritmo es más eficiente que otro por su menor número de comparaciones.

Los algoritmos de ordenamiento se suelen dividir en dos grupos: a) los ordenamientos directos, donde se encuentran el algoritmo de burbúja, algoritmo de selección y algoritmo de inserción; b) los ordenamientos indirectos, donde se encuentran el algoritmo shell, algoritmo de ordenamiento rápido, algoritmo de ordenamiento por mezcla, y el algoritmo radixsort.

4.3.1 Ordenamiento por burbuja

Esté algoritmo es el usado frecuentemente debido a su facilidad para recordarlo e implementarlo, se le llama burbúja o hundimiento debido a que los valores más pequeños gradualmente suben hacia la parte superior del arreglo mientras que los valores más grandes se hunden en la parte inferior del arreglo. El algoritmo consiste en hacer varias pasadas a través del arreglo, y en cada pasada se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente se dejan los valores como están, pero si está en orden decreciente, sus valores se intercambian en el arreglo. En un arreglo de n elementos se necesitan n-1 pasadas para la ordenación.

Algoritmo de Burbuja

```
bandera=1
Repetir desde i=0 hasta que sea menor que n-1 y bandera
        bandera=0
                Repetir desde j=0 hasta que sea  menor que n-i-1
                        Si a[j] > a[j+1]
                        entonces
                                bandera=1
                                aux=a[j]
                                a[j]=a[j+1]
                                a[j+1]=aux
```

Se debe considerar en el algoritmo que los elementos a ordenar se encuentran en el arreglo ***a*** y que dicho arreglo tiene ***n*** elementos. Una alternativa para reducir la cantidad de pasadas, puede ser la siguiente.

Algoritmo de Burbuja Mejorado

```
i=n-1
Mientras i>0
        indice_inter=0
        Repetir de j=0 hasta menor que i
                Si a[j+1] = a[j]
                        aux= a[j]
                        a[j] = a[j+1]
                        a[j+1] = aux
                        indice_inter=j
        i= indice_inter
```

4.3.2 Ordenamiento por selección

El ordenamiento por selección se encarga de colocar los elementos del arreglo en forma ascendente. Se encarga de intercambiar el elemento más pequeño del arreglo con el primer elemento del arreglo. Después de terminar la primera pasada se tiene que en a[0] se encuentra el menor, y los restantes a[1], … , a[n-1] permanecen desordenados. La siguiente pasada se encarga de colocar el elemento más pequeño en a[1], y los restantes a[2], … , a[n-1] permanecen desordenados, pero ya se tiene que a[0] y a[1] están ordenados. El proceso continuará n-1 pasadas y con ello se tendrá que los elementos del arreglo se encuentran ordenados.

Algoritmo de selección

```
Repetir de i=0 hasta menor que n-1
        indice_menor=i
        Repetir de j=i+1 hasta menor que n
                Si a[j] < a[indice_menor]
                        indice_menor=j
        Si i ≠ indice_menor
                aux=a[i]
                a[i]= a [indice_menor]
                a[indice_menor]=aux
```

4.3.3 Ordenamiento por inserción

El ordenamiento consiste en colocar y abrir un espacio donde el elemento deba acomodarse según su orden. Se inicia considerando que el elemento a[0] esta ordenado por ser un sólo elemento. El siguiente elemento a[1] se deberá insertar antes o después de a[0] dependiendo si es mayor o menor. Después el elemento a[2] y se explora a[1] y a[0], y se inserta dependiendo de su orden moviendo hacia abajo una posición los elementos mayores para dejar vacia la posición. Este ciclo se realiza para cada elemento a[i] y explorando a[i-1] hasta a[0].

Algoritmo de inserción

```
Repetir de i=1 hasta menor que n
        j=i
        aux=a[i]
        Mientras j>0 AND aux<a[j-1]
                a[j]=a[j-1]
                j=j-1
        a[j] = aux
```

Anexo 1. GCC y el software libre

Todo lo que queremos es buen software, pero ¿qué significa para nosotros que un software sea bueno?. Funcionalidades adecuadas y fiabilidad puede ser algo *técnicamente* bueno, pero esto no es suficiente. Un buen software debe además ser *éticamente* bueno: tiene que ser respetuoso con la libertad de los usuarios.

Como usuario de software, se debería tener el derecho a ejecutarlo como se necesite, el derecho a estudiar el código fuente y a cambiarlo como se desee, el derecho a redistribuir copias de éste a terceras personas, y el derecho a publicar versiones modificadas con lo que se puede contribuir a la construcción de la comunidad. Cuando un programa respeta la libertad de esta forma, se dice que es *software libre*. Anteriormente a GCC había otros compiladores para C, Fortran, Ada, etc. Pero no eran software libre, y no se podían usar libremente. Escribí el GCC para que se pueda usar un compilador sin perder nuestra libertad.

Un compilador solo no es suficiente —para usar un sistema de computación, se debe disponer de un sistema operativo completo. En 1983, todos los sistemas operativos para ordenadores modernos eran no libres. Para remediar esto, en 1984 comencé a desarrollar el sistema operativo GNU, un sistema similar a Unix que sería software libre. El desarrollo de GCC fue una parte del desarrollo de GNU.

A principios de los 90, el recién terminado sistema operativo GNU fue completado con la suma de un kernel, Linux, que se hizo software libre en 1992. El sistema operativo combinado GNU/Linux ha alcanzado la meta de hacer posible el uso de una computadora en libertad. Pero la libertad nunca está automáticamente asegurada, y debemos trabajar para protegerla. El Movimiento del Software Libre necesita tu apoyo.

Richard M. Stallman Febrero de 2004

https://www.davidam.com/docu/gccintro.es.html

Anexo 2. Secuencias de escape

Secuencia de escape	Descripción
\n	Indica nueva línea. Se encarga de colocar el cursor al principio de la siguiente línea.
\t	Indica tabulador horizontal. Se encarga de mover el cursor al siguiente tabulador horizontal.
\v	Indica tabulador vertical. Se encarga de mover el cursor al siguiente tabulador vertical.
\r	Indica retorno de carro. Se encarga de colocar el cursor al inicio de la línea actual.
\a	Indica alerta. Se encarga se sonar la campana del sistema.
\\	Indica diagonal invertida. Se encarga de imprimir un carácter de diagonal invertida.
\”	Indica doble comilla. Se encarga de imprimir un carácter de doble comilla.
\’	Indica simple comilla. Se encarga de imprimir un carácter de comilla simple.
\b	Indica retroceso. Se encarga de colocar el cursor en un espacio atrás.
\o	Indica constante octal. Se encarga de mandar a imprimir el valor en formato octal.
\x	Indica constante hexadecimal. Se encarga de mandar a imprimir el valor en formato hexadecimal.

Anexo 3. Algunas funciones matemáticas declaradas en math.h

En la siguiente tabla se muestran algunas funciones básicas declaradas dentro de la librería **math.h**.

Función	Descripción
double **sin** (double x)	Calcula el seno del parámetro **x** de tipo double, el resultado se retorna como tipo double.
double **cos** (double x)	Calcula el coseno del parámetro **x** de tipo double, el resultado se retorna como tipo double.
double **tan** (double x)	Calcula la tangente del parámetro **x** de tipo double, el resultado se retorna como tipo double.
double **asin**(double x)	Calcula el arco seno del parámetro **x** de tipo double. $\sin^{-1}(x)$ en el rango $[-\prod/2, \prod/2]$, $x \in [-1,1]$, el resultado se retorna como tipo double.
double **acos**(double x)	Calcula el arco coseno del parámetro **x** de tipo double. $\cos^{-1}(x)$ en el rango $[0, \prod]$, $x \in [-1,1]$, el resultado se retorna como tipo double.
double **atan**(double x)	Calcula el arco tangente del parámetro **x** de tipo double. $\tan^{-1}(x)$ en el rango $[-\prod/2, \prod/2]$, $x \in [-1,1]$, el resultado se retorna como tipo double.
double **atan2**(double y, double x)	Calcula el arco tangente del parámetro **x** de tipo double. $\tan^{-1}(y/x)$ en el rango $[-\prod, \prod]$, el resultado se retorna como tipo double.
extern double **exp** (double x)	Calcula **e** elevada a la potencia **x** (e^x) que es de tipo double, el resultado retornado es de tipo double.
extern double **log** (double x)	Calcula el logaritmo natural de un número **x** de tipo double, y el resultado obtenido lo retorna en un tipo double.
extern double **log10** (double x)	Calcula el logaritmo base 10 del parámetro **x** de tipo double, y el resultado retornado es de tipo double.
extern double **hypot** (double x, double y)	Calcula la hipotenusa a partir de los dos números, **x** y **y**, de tipo double, el resultado se retorna en tipo double.
extern double **pow** (double x, double y)	Calcula el valor de un número **x** de tipo double al elevarlo a la potencia **y** que es un número de tipo double, el resultado retornado es de tipo double.
extern double **sqrt** (double x)	Calcula la raíz cuadrada del número **x** de tipo double, y retorna el resultado en un tipo double.

Anexo 4. Librerías estándares

Nombre de la librería	Contiene	Algunas funciones que contiene
stdio.h	Funciones, macros y tipos para manipular datos de entrada y salida estándar.	printf(); scanf(); getchar(); putchar(); putc (); perror(); fprintf(); fscanf(); fopen(); fclose(); fgetc(); fgets(); fread(); fseek(); fwrite(); fflush(); ferror(); feof(); remove(); rename(); rewind(); setbuf(); sprintf(); sscanf(); ...
stdlib.h	Funciones, macros y tipos para uso general, y en especifico búsqueda y ordenamiento.	atof(); atoi(); atol(); bsearch(); div(); free(); getenv(); malloc(); rand(); srand(); system(); exit(); ...
string.h	Funciones y macros para el manejo de caracteres.	strcpy(); strcat(); strcmp(); strchr(); strrncopy(); strncat(); strncmp(); strstr(); strpbrk(); strerror(); memcpy(); memmove(); memcmp(); memchr(); memset(); ...
math.h	Funciones, macros y tipos de matemáticas.	sin(); cos(); tan(); exp(); log(); log10(); log2(); fabs(); hypot(); pow(); ...
ctype.h	Funciones y macros para clasificación de caracteres.	tolower(c); toupper(c); isdigit(c); isalpha(c);isalnum(c); iscntrl(c); isxdigit(c); islower(c); issupper (c); ispunct(c); isprint(c); ...
errno.h	Funciones donde se definen constantes para los códigos de error	EPERM, ENOENT, ESRCH, EINTR, EIO, ENXIO, E2BIG, ENOEXEC, EBADF, ECHILD, ...
limits.h	Función para definir los límites de los diferentes tipos de enteros.	SCHAR_MAX, SCHAR_MIN, INT_MAX, INT_MIN, LONG_MAX, LONG_MIN, ...
float.h	Función para definir los límites de los tipos punto flotante.	FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP, ...
signal.h	Funciones para el manejo de señales del sistema.	signal(); raise(); exit();
time.h	Funciones, macros y tipos para el manejo de tiempo y fecha del sistema.	asctime(); clock(); ctime(); difftime(); gmtime(); localtime(); mktime(); time(); ...
setjmp.h	Funciones para manipular el entorno al realizar llamadas.	longjmp(); setjmp();

Nombre de la librería	Contiene	Algunas funciones que contiene
curses.h	Funciones y macros para la manipulación del modo gráfico.	clear(); box(); border(); color_set(); erase(); getch(); getstr(); init_color(); ...
unistd.h	Funciones y macros para la manipulación de procesos.	fork(); getpid(): getppid();

Anexo 5. Librería ncurses

ncurses es una librería que permite el manejo de la pantalla en los sistemas UNIX, LINUX y otros sistemas operativos. Permite crear una interfaz de usuario en modo texto. En caso de no tener instala la librería necesita realizarlo a través de las siguientes ordenes, en el sistema operativo Linux distribución Ubuntu:

sudo apt-get install libncurses5-dev libncursesw5-dev

En sistema operativo Linux distribución Fedora:

sudo yum install ncurses-devel

o en distribuciones 22.x

sudo dnf install ncurses-devel

Referencias

[1] Kernighan, B.W., Ritchie, D.M. El lenguaje de programación C. Pearson Educación. 1991.
[2] Haskins, D. C Programming in Linux. Book Boon. 2009.
[3] Deitel, H. M., Deitel, P. J. Como programar en C/C++. Prentice-Hall.
[4] Arroyo, D. Una introducción a GCC. 1995. https://www.davidam.com/docu/gccintro.es.html. Fecha de consulta 2 de septiembre 2021.
[5] Resnick, R., Halliday, D., y Krane, K. Física Volumen 1. Patria. 2001.