Notas de C

Logan Martinez

20 de noviembre de 2020

Resumen

Pues estas son mis notas para aprender lenguaje de progamación 'C' desde "cero". Estas notas están respaldadas en un repositorio de Github.

1. Introducción

1.1. ¿Que es C?

Es un lenguaje de programacionde medio nivel. ya que dispone de estrtucturas que son entendibles a simple vista como los lenguajes de alto nivel, pero tambien permite un control a bajo nivel. ya que permite controlar con facilidad disporitivos perifericos y optimizar el uso de memoria.

1.2. Historia de C

Desarrollado por Dennis Ritchie entre 1969 y 1972 en los laboratorios bell como evolución a su antecesor el lenguaje 'B' y basado en el lenguaje "BCPL".

1.3. Caracteristicas destacables

-Eficacia
-Potencia
-Eficiencia
-Rapidez

1.4. ¿Que temas contendrá el curso?

-Tipos de dato
-Converciones
-Bucles
-Palabras reservadas
-funciones
-Asignacion de memoria
-Directivas
-Pilas
-Colas
-Arboles
-Procesos
-Creacion de libreria

2. Instalación y configuracion de entorno

2.1. Configuracion

Necesitamos GCC, pero ¿Que es GCC? GCC es una coleccion de compiladores del proyecto GNU.

Anteriormete GCC solo compilaba para C, pero posteriormente se extendio para C++, fortran, ada, objetics C, etc.

3. Hola mundo

3.1.

Las lineas que comienzan con este simbolo (#) son procesadas por el preprocesador antes de que el programa se compile.

3.2. Caracter de escape

La diagonal invertda se le conoce como un caracter de escape. Cuando una diagonal invertida se encuentra dentro de una cadena de caracteres, el compilador lo verifica y lo convierte en usa secuencia de escape.

3.3. Practica

- Ingresar a cdm
- entrar a la carpeta utilizando 'cd' seguido de la direccion de la carpeta
- escribir en la consola "gcc (Nombre del archivo).c −o (nombre del archivo).exe"
- escribir en la consola 'dir'
- escribir en la consola '(nombredel archivo).exe'

4. indef

4.1. Operadores

- (&) es un operador de dirección. por lo que indica la variable en la que se va a guardar esa información.

4.1.1. Operadores Aritmeticos

- suma (+)
- resta (−)
- multiplicacion (∗)
- división (/)
- modulo (%)

4.1.2. Operadores de igualdad

- == 'x' es igual que 'y'
- != 'x' es diferente que 'y'

4.1.3. Operadores de relacion

- > 'x' es mayor que 'y'
- < 'x' es menor que 'y'
- >= 'x' es mayor o igual que 'y'
- <= 'x' es menor o igual que 'y'

4.2. Operador condicional

- interrogación (?) es similar al if-else

4.3. Tipos de variable

(%) se utiliza para indicar el tipo de variable.

- %d significa que es una variable entera decimal.
- %i significa que es una variable entera.
- %c significa que es un caracter.
- %li significa entero largo.
- %.'numero de decimales'f significa numero real.
- %.'numero de decimales'lf significa real largo.
- %s Significa cadena de caracteres.

4.4. Variables

- **char** (caracter) es de tipo alphanumerico (%c)
- **int** (entero) (%i)
- **short** (entero corto) (%i)
- **unsigned int** (entero positivo) (%i)
- **long** (entero largo) (%li)
- **float** (real) (%f)
- **double** (real con doble de rango) (%f)

4.5. Condicional

- Se ultiliza if (condición)
  {
  /*instruccion*/
  }
- else {
  /*instruccion*/
  }

5. Directivas del pre-procesador

Las librerias del pre-procesador son las que contienen librerias y macros. Todas las directivas comienzan con un simbolo de numeral (#).

5.1. *Include*

- **#include** <stdio.h>
  #include "(nombre del archivo)"

La diferencia entre ambas es la ubicacion en que el pre-procesador busca el archivo a incluir.

Si el nombre del archivo se encierra entre comillas, el pre-procesador busca el archivo a incluir en el mismo directorio donde se encuentra el archivo que va a compilarse.

Si el nombre del archivo se encierra entre llaves angulares va a buscarlos en los encabezados de la biblioteca estándar.

- math.h
- stdio.h
- stdlib.h
- time.h
- etc.

5.2. *Define*

La directiva **#define** crea constantes simbólicas y macros.

Ejemplos:

- PI 3.14159
- CUBO(a) a*a*a

6. Palabras reservadas y conversion de tipos de dato

6.1. Palabras reservadas

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

6.2. Conversion de tipos de dato

Para cambiar de tipo de dato se escribe.

...
printf("datos\n",variable, (tipo de dato)variable, (tipo de dato)variable, ...);

7. Funciones I

El concepto del lenguaje C esta basado en Bloques de construcción, estos bloques se llaman *funciones*. un programa en C es una colección de 1 o mas funciones. En C una función es una subrutina que contiene una o mas sentencias y hace una o mas tareas. En un bien hecho programa en C, cada función hace una sola tarea.

En general puedes dar a una función el nombre que quieras, a excepción de ***main*** que está reservada para la función que inicia la ejecución de tu programa.

7.1. Función con argumentos

Una función argumento es simplemente un valor que es pasado a la función al mismo tiempo que este es llamado.

Ejemplo:

```
#include <stdio.h>
void sqr(int x)
{
printf(" %d cuadrado es %d\n",x,x*x);
}
int main(){
int num;
num=100;
sqr(num);
return 0;
}
```

7.2. Funciones que retornan valores

```
#include <stdio.h>
int mul(int a, int b)
{
return a*b;
}
int main()
{
int answer;
answer = mul(10,11);
printf(" %d\n",answer);
return 0;
}
```

7.3. Forma general de una funcion

return-type function-name (*parameter list*)
{
body of code
}

Para funciones sin parámetros, no habrá lista de ellos.

8. Ciclos

Cuando empezamos a hablar de ciclos hacemos referencia a que tendremos un mejor control del programa. La mayoría de los programas involucran un ciclo donde podemos tomar decisiones pero un poco mas controlado.

8.1. Ciclo *for*

Ejemplo:

```
...
for(i = 1;i <= 10;i++)
{
/*instrucciones*/
}
```

8.2. Ciclo *while*

Nos permite especificar una acción mientras la condición sea verdadera.

Ejemplo:

```
...
int i=1;
while(i <= 10)
{
/*instrucciones*/
i++;
}
```

8.3. Ciclo *Do while*

Ejemplo:

```
...
int i=1;
do {
/*instrucciones*/
i++;
}while(i <= 10);
```

9. *Switch*

Consiste en un grupo de etiquetas **case** y un caso opcional **default** que nos va a permitir tener el control dependiendo de los casos que haya en nuestro programa.

9.1. Etiqueta case

Aquí se escribe el nombre del caso; Si es algún carácter debe ir entre comillas o si es un numero puede ser escrito sin ellas.

Ejemplo:

```
...
int casos;
printf("ingresa un numero/n");
scanf(" %i", &casos);
```

```
switch(casos)
{
case 1:
printf("elegiste el primer caso/n");
break;

case 2:
printf("elegiste el segundo caso/n");
break;

case 3:
printf("elegiste el tercer caso/n");
break;

case 4:
printf("elegiste el cuarto caso/n");
break;

case 5:
printf("elegiste el quinto caso/n");
break;

default:
printf("no encontramos tu caso/n");
break;
}
```

10. Arreglos

Los arreglos son un conjunto de datos que se van almacenando dentro de una variable. Los arreglos nos permiten guardar muchos de estos datos; en estos casos los arreglos son conocidos como una unidad estática, ya que estos no cambiaran su tamaño durante la ejecución del programa.

10.1. Arreglo unidimensional

Ejemplo:

```
...
int sizeA;

printf("tamaño del \n");

scanf(" %i",&sizeA);
int age[sizeA];
for(int i = 0;i < sizeA;i++)
{
printf("ingresa el valor %\n",i+1);
scanf(" %i", &age[i]);
}
printf("los valores del arreglo son:\n");
for (int i = 0; i < sizeA;i++)
{
printf(" %i-", age[i]);
}
printf("\n");
```

10.2. Strings

Por mucho el uso mas común de un arreglo unidimensional es para crear cadenas de caracteres. La forma mas sencilla de ingresar una cadena desde el teclado es con la funcion ***gets()***

Para ejemplos de funciones para strings ...\learn c\c string_stuff.c

10.3. Arreglo multidimensional

Ejemplo:

```
...
/*
col.. 8 1 2
fila0 5 3 1
fila1 6 4 2
*/
int multi[2][3] = {{5,3,1},{6,4,2}};
printf(" %i\n",multi[0][2]);
```

Se pueden crear arreglos sin especificar el tamaño, en ese caso C se encarga de crear un arreglo lo suficientemente grande para mantener los datos en el.

Ejemplo:

```
char el[ ] = invalid imput;
```

11. *Break* y *Continue*

Nos permiten manejar el flujo de nuestro programa como queramos.

Estos dos no son considerados como parte de la programación estructurada; sin embargo nos pueden sacar de un gran apuro en ocasiones.

- **break** termina la ejecución de algún ciclo.
- **continue** nos permite seguir ejecutando, pero salta un paso.

11.1. *Break*

Ejemplo:

```
...
int num =0;
while(num<=7)
{
if(num == 2)
{
break;
}
printf(" %i\n",num);
num++;
```

En este caso se detiene el programa.

11.2. *Continue*

Permite ejecutar o continuar nuestro ciclo porque aun hay valores.
Siempre se aumenta el valor antes del if, sino se cicla el programa y se detiene.

Ejemplo:

```
...
int num =0;
while(num<=7)
{
if(num == 2)
{
break;
}
printf(" %i\n",num);
num++;
while(num<=7)
{
if(num == 2)
{
continue;
}
printf(" %i\n",num);
```

En este caso se salta el imprimir el numero 2.

12. Funciones II

12.1. Funciones básicas

Ejemplo saludo:

```
...
void saludo()
{
printf("Hola mundo\n");
return;
}
int main(){
saludo();
return 0;
}
```

Ejemplo suma:

```
int suma();
int main();
{
printf(" %i\n",suma());
return 0;
}
int suma ()
{
int num1 =12;
int num2 =4;
int suma = num1 + num2;
```

```
return suma;
}
```

12.2. funciones de retorno

Ejemplo:

```
...
int suma();
int num3= 2;
int main();
{
int num1,num2;
printf("ingresa el primer valor\n");
scanf(" %i",&num1);
printf("ingresa el segundo valor\n");
scanf(" %i",&num2);
printf(" %i\n",suma(num1,num2));
return 0;
}
int suma (int num1, int num2)
{
int suma = num1 + num2 + num3;
return suma;
}
```

12.3. Funciones re-cursivas

Son funciones que tienen la propiedad de llamarse a si mismas.

Ejemplo del factorial de un número:

```
...
/*
5! = 5*4*3*2*1 o 5*4!
4! = 4*3*2*1 o 4*3!
3! = 3*2*1 o 3*2!
2! = 2*1 o 2*1!
1! = 1
0! = 1
*/

long Factorial(long numero);
int main()
{
int numero;
printf("ingresa un número\n");
scanf(" %i",&numero);
for (int i = 0; i <= numero; ++i)
{
printf(" %ld\n",Factorial(i));
}
return 0;
}
long Factorial(long numero)
{
```

```
if(numero <= 1)
{
return 1;
}else{
return(numero * Factorial(numero-1));
}
}
```

12.4. Forma general de una función

La forma profecional de una funcion es:

type-specifier function_name(parameter declarations)
{
body of the function
}

El tipo de la función especifica el valor que retorna usando la sentencia **return**. Puede ser de cualquier tipo valido.La lista de declaración de parámetros es una lista de variables, con sus respectivos tipos y nombres separadas por comas que recibirán los valores de los argumentos cuando al función es llamada. Una función puede no tener parámetros, sin embargo aun son necesarios los paréntesis.

Es importante entender que a diferencia de las variables, los parámetros deben de siempre incluir nombre y tipo. La forma correcta de hacerlo es por ejemplo:

```
f(int x, int y, float z){}
```

12.5. Sentencia return

La sentencia **return** tiene 2 importantes usos. Primero causará una salida inmediata de la función actual. Segundo, se puede usar para retornar un valor.

12.5.1. Retornando desde una función

Hay 2 maneras en las que una función termina la ejecución y retorna al llamador. La primera manera es cuando la ultima sentencia de la función es ejecutada y se encuentra con el final de la función. Por ejemplo, esta funcion simplemente imprime en sentido contrario en la pantalla:

```
void pr_reverse(char *s)
{
register int t;
for(t = strlen(s)-1; t > -1; t–) printf(" %c", s[t]);
}
```

una vez la cadena es desplegada, la función no tiene mas que hacer, entonces regresa a el lugar desde que fue llamada.

la segunda forma una en que funcion puede regresar es desde el uso de la sentencia **return**. La sentencia **return** puede ser usado sin algun valor asociado a el. Por ejemplo:

```
void power(int base, int exp)
{
int i;
if(exp ¡0) return; /*Cant do negative exponents*/
i = 1;
for( ; exp; exp–) i = base*i;
printf("the answer is: %d: ", i);
}
```

12.5.2. Retornando un valor

Para retornar un valor desde una función, deberías seguir la sentencia **return** con el valor que será retornado. Por ejemplo, esta función retorna el valor máximo de sus 2 argumentos:

```
max(int a, int b)
{
int temp;
if(a > b) temp = a;
else temb = b;
return temp;
}
```

Es posible que una función contenga 2 o mas sentencias **return**. Mas de un retorno es seguido usado para simplificar y hacer mas eficiente algún algoritmo. Por ejemplo:

```
max(int a, int b)
{
if(a > b) return a;
else return b;
}
```

Es importante tener en cuenta que tener múltiples **return** puede des-estructurar una función y hacer que su significado sea confuso. El mejor consejo es usar múltiples **return** solo cuando contribuyan de manera significativa al rendimiento de la función. todas las funciones, excepto las tipo **void** retornan un valor. Este valor está explícitamente especificada por la sentencia **return**.

Como todas las funciones, excepto las tipo **void** retornan valores, cuando escribes programas, tus funciones generalmente serán de 3 tipos. El primero es simplemente computacional. Está específicamente diseñado para realizar operaciones en sus argumentos y retornan un valor basado en esa operación, en esencia es una función "pura". Ejemplos de esto son la función **sqrt()** y **sin()**, que retornan el un numero raiz cuadrada y seno, respectivamente.

El segundo tipo de funciones manipulan información y retorna un valor indicando el éxito o fallo de esa manipulación. Un ejemplo es **fwrite()**, que es usado para escribir información a un archivo de disco. Si la operación de escritura es exitosa **fwrite()** retorna el numero de cosas pediste sean escritos; cualquier otro valor, indica que un error ha ocurrido.

El ultimo tipo de función no tiene un tipo explicito de valor de retorno. en esencia, la función es estrictamente procedimental y no produce valor. por turbias razones históricas, muchas veces funciones que realmente no producen un resultado interesante, retornan algo igualmente. Por ejemplo, **printf()** retorna el numero de caracteres escritos; seria muy inusual encontrar un programa que de hecho revise esto. Por lo tanto , aunque todas la funciones (excepto aquellas tipo **void**) retornan algo, no necesariamente tienes que utilizarlos para algo.

12.6. Funciones retornando valores no enteros

Cuando es necesario retornar diferentes tipos de datos a **int**se requiere un proceso de 2 pasos. Primero declarar la función; segundo, se le debe decir al compilador el tipo de la función antes de su primera llamada.

Las funciones pueden ser declaradas para retornar cualquier tipo de dato valido en C. El método de declaración es similar a la de variables: el tipo especificado precede el nombre de la función, el especificador de tipo le dice al compilador que tipo de dato la función retorna. La mejor manera de informar al compilador sobre el tipo de retorno de una función es el usar un prototipo de función

12.6.1. Usando funciones prototipo

Una función prototipo realiza 2 tareas. Primero identifica el tipo de retorno de una función. Segundo especifica en tipo y numero de argumento que una función recibe. Tiene la siguiente forma general:

type function-name(parameter list);

El parámetro por convención va cerca de la parte superior del programa o en un header file y debe ir antes de cualquier llamada se hace de la función.

12.6.2. Retornando punteros

Como cada tipo de dato puede tener una longitud diferente, el compilador debe saber que tipo de dato el apuntador está apuntando para apuntar al siguiente dato. Por lo tanto, una función que retorna un puntero debe ser declarado como ello. Por ejemplo aquí hay una función que retorna un puntero a una cadena en el lugar donde un carácter que coincide es encontrado:

...\learn c\c pointer_return.c

12.6.3. Funciones de tipo void

cuando una función no retorna un valor, puede ser declarada como tipo **void**. hacerlo evita su uso en cualquier expresión y ayuda a evitar errores accidentales. Por ejemplo:

...\learn c\c function_void.c

12.7. Mas en prototipos

12.7.1. Desajustes de argumento

Ademas de decir al compilador sobre el tipo de retorno de una función, un prototipo también previene a la función de ser llamada con un tipo incorrecto o numero de argumentos. aunque C automáticamente convertirá el tipo de un argumento a el tipo del parámetro que está recibiendo, algunos tipos de conversión son simplemente ilegales. Por ejemplo:

...\learn c\c illegal_argument.c

Ademas, el usar prototipos de función, ayuda a encontrar bugs antes de que ocurra, reviniendo a una función el ser llamada con argumentos inválidos, también ayuda a verificar que tu programa trabaja correctamente, no permitiendo llamar funciones con el numero equivocado de argumentos.

12.7.2. Archivos de cabecera

Los archivos de cabecera contienen 2 principales cosas: ciertas definiciones usadas por las funciones y los prototipos de las funciones estándar relacionadas con el archivo de cabecera.

12.7.3. Forma general prototipos de función

La forma general de los prototipos de función es:

tipo nombre(parametros);

En C es posible escribir un prototipo parcial de la funcion, evitando escribir los parametros, sin embargo esto no es valido para C++, por lo que se recomienda siempre evitar escribir un prototipo parcial de la funcion.

12.8. Reglas de alcance

Las reglas de alcance de un lenguaje son las reglas que gobiernan ya sea una pieza de código que sabe o tiene acceso a otra pieza de código o dato.

En C, cada función es un bloque discreto de código. El código de una función es privado a otra función y no puede ser accesado por alguna sentencia en cualquier otra función, excepto por una llamada de esa función. El código que compromete el cuerpo de una función está escondido del resto del programa y a no ser que use variables o datos globales, no puede afectar ni ser afectado por otras partes del programa, excepto a la especificación de tu programa. Visto de otra manera, el código y los datos que están definidos en una función, no pude interactuar con el código o los datos definidos en otra función, a no ser sea explícitamente especificado, porque las 2 funciones tienen diferente alcance.

Hay 3 tipos de variables: *variables locales, parámetros formales* y *variables globales*. Las reglas de alcance gobiernan como cada una de estas puede ser accesado por otras partes de tu programa u establece.

12.8.1. Variables locales

Las variables que son declaradas dentro de una función son llamadas *variables locales*. Una variable puede ser declarada dentro de cualquier bloque de código y ser local en el. Lo mas importante de entender sobre variables locales es que exixten solo mientras el bloque de cofigo en el que fueron declarados se extá ejecutando. el bloque de codigo mas comun en el que las variables locales son declaradas es la funcion. Pir ejemplo:

...\learn c\c local_variables.c

12.8.2. Parámetros formales

Si una función usa argumentos, entonces deberían declararse variables que que acepten los valores de esos argumentos, estas se conforman como cualquier otra variable local, esto son los parámetros formales.

12.8.3. Variables globales

Las variables locales son conocidas durante toda la ejecución del programa, por lo que se pueden utilizar en cualquier bloque de código en el programa, como consecuencia de esto, mantienen su valor mientras no se le hagan asignaciones.

Cuando una variable global y una variable local tienen el mismo nombre, todas las referencias a una variable con ese nombre dentro de la función donde se declara la variable local, se referirán a esa variable local. Ademas no afectará a la variable local, esto puede ser beneficioso, sin embargo puede hacer que tu programa actúe extraño aunque parezca estar bien.

Deberías evitar usar variables globales innecesarias por 3 razones:

- Toman memora todo el tiempo en que tu programa se ejecuta, no solo cuando se necesita.
- Usando un global donde una variable local hará a una funcion menos general, pues depende en algo que se define fuera de si.
- Usar un gran numero de variables globales pueden llevar al programa errores por desconocidos y desagradables efectos secundarios.

Esta ultima razón esta evidenciada en BASIC, donde todas las variables son globales. Un grave problema desarrollando largos programas es accidentalmente cambiando el valor de una variable porque fue utilizada demasiadas variables globales en tu programa.

Uno de los principales puntos de un lenguaje estructurado es la compartimentación de código y datos. En C compartimentación es lograda a través de el uso de funciones y variables locales. Por ejemplo:

- Especifico

  int x, y;
  mul()
  {
  return (x*y);
  }

- General

  mul(int x, int y)
  {
  return (x*y);
  }

Ambas funciones retornarán el producto de variables **x** y **y**. sin embargo, la versión generalizada o *parametrizada* puede ser utilizada para retornar el producto de cualquier par de numeros, mientras que la version especifica puede ser utilizada para encontrar solo el producto de las variables globales **x** y **y**.

12.9. Funciones, parámetros y argumentos

12.9.1. Llamada por valor, llamada por referencia

En general, subrutinas pueden ser pasados por argumentos en una de dos maneras. El primer método es llamado *llamada por valor*. Este método copia el valor de un argumento en el parámetro formal de la subrutina. Por lo tanto, cambios hachos en el parámetro de la subrutina no tiene efectos en la variable usada para llamarla.

Llamada por referencia es la segunda manera en que a una subrutina se le puede pasar un argumento, la dirección de un argumento es copiado en el parámetro. Dentro de la subrutina, la dirección es usada para acceder al propio argumento usado en la llamada. esto significa que los cambios hechos a el parámetro afectara a la variable usada para llamar la rutina.

12.9.2. Creando una llamada por referencia

Aun cuando la convención en C para pasar parámetros es la llamada por valor, es posible crear una llamada por referencia pasando un puntero como argumento. Esto hace posible cambiar el valor de el argumento fuera de la función. Por ejemplo:

...\learn c\c pointers_everywhere.c

...\learn c\c llamada_referencia.c

En este punto deberías de haber podido entender porque tienes que poner **&** enfrente de los argumentos para **scanf()**. De hecho lo que estas haciendo es pasar su dirección para que la variable llamada pueda ser modificada.

12.9.3. Llamado funciones con arreglos

Cuando un arreglo es pasado como argumento de una función, solo la dirección del arreglo se pasa, no se copia todo el arreglo. Cuando tu llamas una función con un arreglo nombre, un puntero a el primer elemento del arreglo se pasa a la función (recuerda, en C el nombre de un arreglo sin algún indice es un puntero al primer elemento de ese arreglo.) Esto significa que la declaración del parámetro debe ser compatible con un tipo puntero. Por ejemplo:

...\learn c\c pointer_return.c

...\learn c\c llamada_arreglo.c

Esto está permitido porque cualquier puntero puede ser indexado como si fuera un arreglo.

Hay que reconocer que los 3 métodos de declarar un parámetro arreglo llevan al mismo resultado: un puntero.

Un arreglo elemento usado como un argumento e tratado como cualquier otra variable. Por ejemplo:

...\learn c\c array_element_pointer.c

12.10. Argumentos de main

La forma general de pasar información a **main()** es a través de el uso de argumentos de la linea de comandos. Esto es la información que va después del nombre del programa(y su extensión).

Hay 3 solamente maneras de construir argumentos a main. Los primeros 2 son **argc** y **argv** son usados para recibir argumentos en linea de comandos. El tercero es **env** y es utilizado para accesar a los parámetros activos ambientales DOS en el momento en que el programa comienza su ejecución.

argc() almacena el numero de argumentos en la linea de comandos y es un entero, siempre será al menos 1, Porque el nombre del programa califica como primer argumento. El parametro **argv** es un puntero a un arreglo de cadenas. Todos los argumentos de linea de comandos son cadenas, cualquier numero tendrá que ser convertido por el programa al formato correcto. Por ejemplo:

...\learn c\c main_argument.c

Cada argumento de comando de linea debería ser separado por un espacio o un tab, comas, semi-columnas y similares no son considerados separadores.

Se pueden ejecutar series de comandos DOS en la linea de comando. Esto se logra usando la función de librería **system()**. Por ejemplo:

...\learn c\c main_arg_system.c

El parámetro **env** es declarado igual que el parametro**argv**. Este es un puntero a un arreglo de cadenas que contienen las configuraciones ambientales.

12.11. Retornando valores desde main()

Cuando retornas una valor entero desde **main()** se pasa al proceso de llamado, usualmente al sistema operativo. Para DOS y OS/2, un retorno de valor 0 indica una terminación exitosa del programa, cualquier otro valor indica que el programa termino debido a un error. Por ejemplo:

...\learn c\c return_error_main.c

12.12. recursion

En C y C++, las funciones pueden llamarse a si mismo. una función es re-cursiva si un estatuto en el cuerpo de la función se llama a si misma. cuando una función se llama a si mismo, nuevas variables locales y parámetros son guardados en el stack y el código de la función es ejecutada con estas nuevas variables desde el inicio. Una llamada re-cursiva no hace una copia de la función, solo se hacen nuevas variables y parámetros.

la mayoría de las rutinas re-cursivas, no reducen significativamente el tamaño del código y el almacenamientos de variables, ademas de que la mayoría suelen ser un poco mas lentas que sus equivalentes iterativos, por la re-llamada de la función, sin embargo no es notable en la mayoría de los casos. Algunas funciones pueden causar *stack overrun*, pero esto no es usual que pase yque cause un crash.

la principal ventaja de funciones re-cursivas es que pueden usarse para crear mal limpias y simples versiones de varios algoritmos que sus hermanos iterativos. por ejemplo el algoritmo de ordenamiento *QuickSort* es difícil de implementar de forma iterativa. también algunos problemas relacionados con *AI* tienden a llevar a si mismos a soluciones re-cursiva. Finalmente, algunas personas suelen pensar que las formas re-cursivas son mas fáciles. Por ejemplo:

...\learn c\c recursivas.c

12.13. problemas de implementación

12.13.1. parámetros y funciones de propósito general

Una función de propósito general es la que se utiliza en una amplia variedad de situaciones. Típicamente no deberías basar funciones de propósito general en datos globales, Es mucho mejor que cualquier cosa pasar toda la información una función necesita por sus parámetros.

12.14. Eficiencia

En ciertas especializaciones aplicaciones, quizá necesites eliminar una función y remplazarlo con *codigo in-line*. Esto es equivalente a una sentencia de una función usada sin llamar a esa función.

Existen 2 rezones de porque código in-line es mas rápido qe una función llamada. Primero, una instrucción llamada toma tiempo para ejecutar. Segundo, si hay argumentos para pasar, estos tienen que ser colocados en el stack, que también toma tiempo. Para la mayoría de aplicaciones, esto es un incremento muy bajo, sin embargo, pero cuando es para muy importantes tareas contra el tiempo es critico.

13. Apuntadores

13.1. Conceptos básicos

Los apuntadores son variables cuyos valores son direcciones de memoria. por lo general una variable contiene directamente un valor en especifico; por otro lado un apuntador contiene la dirección de una variable que contiene un valor especifico.

Una variable apuntador se define como:

```
...
int a = 2;
int *apt = &a;
```

Se imprime como:

```
...
printf(“ %i\n”,*apt);
```

Para imprimir la dirección de memoria como un numero hexadecimal se hace:

```
...
printf(“ %p\n”,apt);
```

13.1.1. Aritmética de apuntadores

Solo hay 2 tipos de operadores que deberian ser usados con apuntadores suma y resta de enteros. Cada incremento/decremento en un apuntador se guia por:

ptr = ptr + (sizeof(tipo_de_dato) * n)

13.2. Apuntadores y arreglos

Existe una relación cercana entre arreglos y apuntadores, considera este fragmento.

```
char str[80], *pl;

pl = str;
```

Aquí **pl** ha sido asignado a la dirección del primer elemento del arreglo **str**. otra forma de escribir esto es:

```
pl = &str[0];
```

sin embargo esta es considerada una forma pobre por mayor parte de los programadores de C, si quisieras accesar al quinto elemento en **str** podrias escribir:
str[4]

o

*(pl+4)

Ambas maneras regresarán el quinto elemento.

En C es posible indexar un apuntador como si fuera un arreglo. Esto hace mas evidente la cercana relación entre arreglos y apuntadores. Por ejemplo, este fragmento es completamente valido:

```
int i[5] = {1, 2, 3, 4, 5};
int *p, t;

p = i;

for (t = 0; t < 5; ++t)
{
printf(" %d\n", p[t]);
}
```

En C, p[t] es idéntico a (p+t)

Hasta ahora, los ejemplos han estado concentrados en asignar la dirección de el inicio de un arreglo a un apuntador. Sin embargo, es posible asignar la dirección de un elemento especifico de un arreglo aplicando el **&** a un arreglo indexado, por ejemplo este fragmento pone la dirección del tercer elemento de **x** en **p**:

```
p = &x[2];
```

Un lugar donde esta practica es especialmente útil es en encontrar una sub-cadena. Por ejemplo, este programa imprimirá desde que el primer espacio es encontrado. ...\learn c\c array_address.c

13.2.1. Arreglo de apuntadores

Apuntadores pueden ser arreglados así como harías con cualquier otro tipo de datos. La declaración de un arreglo de punteros **int** de tamaño 10 es:

int *x[10];

Para asignar la dirección de una variable entero a el tercer elemento de un arreglo puntero, escribirías

x[2] = &var;

Para encontrar el valor de **var**, escribirías

*x[2]

un uso común de los arreglos punteros es el mantener punteros de mensajes de error. puedes crear una funcion que imprima un mensaje. Por ejemplo:

```
char *err[ ] = {
"cannot open fill\n",
"read error\n",
"write error\n",
"media failure\n"
};

void serror(int num)
{
printf (" %s", err[num]);
}
```

Como puedes ver, **printf()** está dentro de **serror()** con un apuntador carácter, el cual apunta a uno de los varios mensajes de error indexados por el numero de error pasado de una función. Por ejemplo, si a **num** se le pasa 2, entonces el mensaje **write error** es desplegado.

Otra interesante aplicación para los arreglos de punteros inicializados usa la función **system()** de C, que le permite a tu programa mandar un comando al sistema operativo. la llamada de **system()** tiene esta forma general

system("*command*");

Donde *command* es el comando de sistema operativo a ejecutar. Por ejemplo, asumiendo un ambiente DOS, esta sentencia hace que el directorio default sea desplegado.

system("*DIR*");

El siguiente programa implementa un muy pequeño menu-driven user interface que puede ejecutar cuatro DOS comandos: DIR, CHKDSK, TIME Y DATE.

...\learn c\c array_of_pointers.c

Para una mejor ilustración de la inter-coneccion entre arreglos y punteros, desarrollaremos un simple programa traductor ingles a alemán.

...\learn c\c english_german.c

El arreglo **trans** es de hecho un arreglo de punteros de las cadenas bajo su declaración.

13.3. Punteros a punteros

El concepto de arreglos de punteros es directo porque los indices mantienen su significado claro. Sin embargo, punteros a punteros pueden ser muy confusos.

Un puntero a un puntero es una forma de *indireccion multiple*, o cadena de punteros. Como puedes ver en la figura 8-3. (Using turbo C++, pg. 207), en el caso de un puntero normal, el valor de el puntero es la dirección de la variable que contiene el valor designado. En el caso de un puntero a un puntero el primer puntero contiene la dirección de el segundo puntero, el cual apunta a la variable, la cual contiene el valor designado.

Indireccion múltiple puede ser llevada acabo en a manera deseada, pero hay algunos casos donde mas de un puntero a un puntero es necesario, o de hecho deseable usarlo. Excesiva indireccion es difícil de seguir y propensa a errores (no confundas múltiple indireccion ncon *listas linkeadas* que se utilizan en base de datos y similares).

una variable que es un puntero a un puntero debería ser declarado como ello. Esto se hace poniendo un asterisco adicional en frente de su nombre. Por ejemplo:

float **newbalance;

Es importante entender que **newbalance** no es un puntero a un floating-point number, sino un puntero a un puntero float.

para acceder al valor apuntado apuntado indirectamente atravez de un puntero de un puntero, el operador asterisco debe ser aplicado dos veces como se ve en este pequeño ejemplo.

...\learn c\c pointed_pointer.c

13.4. Inicializar punteros

Después de que un puntero es declarado pero antes de asignarle un valor contendrá un valor indefinido. Si intentas usar un puntero antes de darle un valor probablemente crashearas no solo tu programa, sino también tu sistema operativo.

por convención un puntero que apunta a nada debería de darse le un valor null, para dignificar que apunta a nada. Sin embargo, solo por que un puntero tenga un valor null no implica que sea "seguro", si corres un puntero con valor nulo, aun corres el riesgo de crashear tu programa y el sistema operativo.

porque un puntero nulo es asumido que permanecerá desusado, puedes usarlo para hacer muchas de tus rutinas de punteros mas fáciles de programar y mas eficientes. Por ejemplo, puedes usar un puntero nulo para marcar el final de un arreglo de punteros. si esto se hace, una rutina que accese ese arreglo sabrá que se ha alcanzado el final cuando el valor nulo es encontrado. Ejemplo:

```
for(t = 0; p[t]; ++t)
{
if(!strcmp(p[t], name)) break;
}
```

Este es un ejemplo de una practica muy común en programación profesional de C. Otra variacion en este tema es el siguiente ejemplo de la declaracion de una cadena:

char *p = "hello world\n";

como puedes ver el puntero **p** no es un arreglo. La razon que este tipo de inicializacion funciona, tiene que ver con como C maneja cadenas constantes, los compiladores de C crean una tabla de cadenas donde guardan las cadenas constantes usadas por el programa. Por ejemplo:

...\learn c\c null_pointer.c

sin embargo, tu programa no debe hacer asignaciones a la tabla de cadenas atraves de **p**, pues tu programa puede corromperse (NO ASIGNES VALORES NUEVOS AHI).

13.5. Problemas con punteros

Nada te traerá mas problemas que un puntero "salvaje". punteros son un milagro mixto. ellos dan un tremendo poder y son necesarios para muchos programas, pero cuando un puntero accidentalmente contiene un valor equivocado, puede ser el bug mas difícil de encontrar.

Un bug de puntero erróneo es difícil de encontrar porque el puntero en si no es el problema; el problema es que cada vez que intentas perforar una operación usándola, estas leyendo o escribiendo en una pieza desconocida de la memoria. Si tu lees de ella, lo peor que puede pasar el que tengas basura. Sin embargo, si escribes en ella puedes estar escribiendo sobre otras piezas de tu código o datos. Esto puede no mostrarse hasta después en la ejecución del programa y puede dirigirte a buscar el bug en el lugar equivocado. Esto puede dar muy poca a nada de evidencia sugiriendo que el puntero es el problema. Este tipo de bug ha causado a programadores perder tiempo y sueño.

Porque errores de punteros son una pesadilla, deberías dar lo mejor por nunca generar uno. Para hacer la asignacion de una variable **x** a un puntero **p** debe de ser:

p = &x;

o

int *p = &x

Y se imprime:
printf(" %d", *x);

Si es un arreglo puede ser:
p = i;,

o

int *p = i;

o

int *p = &i[0];

Y se imprime:
printf(" %d", *x);

Pero si es una cadena puede ser:
printf(p);

o

printf(" %s", p);

No evites utilizar punteros solo por que cuando son manejadas incorrectamente pueden causar bugs muy engañosos. Deberías ser cuidadoso y asegurarte de que sabes donde cada puntero apunta antes de usarlo.

13.6. Llamadas por referencia

Existen 2 maneras de pasar argumentos a una función.

- llamadas por valor
- llamadas por referencia

Hasta ahora hemos utilizado las funciones y hemos pasado los argumentos por valor, pero muchas funciones requieren la capacidad de modificar una o mas variables en una sola llamada de la función. En ese caso podemos evitar sobrecargas de pasar objetos por valor.
Las sobrecargas en si es hacer copias del objeto o de nuestra variable.

La diferencia es que las funciones se definen de tipo **void**, las cuales no están obligadas a devolver un valor.

Ejemplo:

```
...
void cubo(int *n);
int main()
{
int num = 5;
printf("El valor original es: %i\n",num);
cubo(&num);
printf("El nuevo valor es: %i\n",num);
return 0;
}
void cubo(int *n)
{
*n = *n * *n * *n;
}
```

14. I/O y Archivos

14.1. Streams y archivos

El sistema de C I/O provee un nivel de abstracción entre el programador y el artefacto usado. Esta abstracción es llamada *stram* y el propio dispositivo es llamado *archivo*. Es importante conocer como ellos interactuan.

14.2. Streams

El ANSI C sistema de archivos, está diseñado para trabajar en una gran variedad de dispositivos, incluyendo terminal, disk driver y tape drives. Aunque cada dispositivo es muy diferente, el ANSI C sistema de archivos, transforma cada uno en un dispositivo lógico llamado stream. todos los strams son similares en su comportamiento. Porque strams son ampliamente independientes de los dispositivos, la misma funcion que escribe en un archivo en disco puede también escribir en la consola, Existen 2 tipos de streams: texto y binario.

14.2.1. Streams de texto

Es una secuencia de caracteres. En un stream de texto, ciertos caracteres de traducción pueden ocurrir como requeridos por el ambiente anfitrión. Por ejemplo, una nueva linea puede ser convertida, de manera que el numero de caracteres escritos o leídos son necesariamente los mismos que encontramos en el dispositivo externo.

14.2.2. Streams binarios

Es una secuencia de bytes que tiene una correspondencia 1-a-1 a eso encontrado en el dispositivo externo. Eso es que no se hace traducción de caracteres. El numero de bytes escritos o leídos, será el mismo que el numero de bytes encontrados en el dispositivo externo. Sin embargo, streams binarios pueden estar repletos de bytes nulos para que llene un sector de un disco

14.2.3. Archivos

En C, un archivo es un concepto lógico que puede se aplicado a todo desde archivos de discos a terminales. Un stream está asociado con un especifico archivo, haciendo una operación abrir. Cuando se abre un archivo, información puede ser intercambiada entre archivo y programa.

no todos los archivos tienen las mismas capacidades. Por ejemplo, un archivo de disco puede soportar acceso aleatorio mientras un disco de cinta no puede. Esto marca algo muy importante del sistema de C I/O: todos los streams son iguales, pero no todos los archivos son iguales.

Si el archivo puede soportar acceso aleatorio (a veces llamado *solicitud de posición*), entonces abrir ese archivo también inicia el *indicador de posición de archivo* para empezar del archivo.

14.3. Conceptual contra real

Tan lejos como le concierne al programador, todo I/O pasa a traves de streams, que son las secuencias de caracteres. Todos los streams son lo mismo. el sistema de archivos une un stream con un archivo. En C, un archivo es cualquier dispositivo externo, capaz de I/O.

14.4. Consola I/O

Esto se refiere a operaciones que ocurren en el teclado y monitor de tu computadora.

14.4.1. Leyendo y escribiendo caracteres

Las funciones de consola I/O mas simples son **getche()**, que lee un caracter desde el teclado y **putchar()**, que imprime un caracter en la pantalla. La funcion **getche()** espera hasta que una tecla es presionada y entonces retorna un valor. La tecla presionada es "echoed.ª la pantalla automaticamente. La funcion **putchar()** escribirá un caracter argummento a la pantalla en la posicion actual del cursor. El prototipo de **putchar()** y **getche()** es este.

```
int getche(void);

int putchar(int c);
```

...\learn c\c putchar.c

El archivo de cabecera de **getche()** es CONIO.H y el de **putchar()** es STDIO.H, El archivo de cabecera de la función **islower()** es CTYPE.H. Ejemplo:

...\learn c\c case_switcher.c

Hay 2 importantes variaciones en **getche()**. El primero es **getchar()**, que es la función original basado en UNIX de ingresar caracteres. El problema con **getchar()** es que se cicla hasta que se ingresa un carácter, por ello su uso no es recomendado.

la segunda y mas útil variación de **getche()** es **getch()**, que opera precisamente como **getche()**, excepto que el carácter no es impreso en la pantalla.

14.4.2. Leyendo y escribiendo cadenas

El siguiente paso en términos de complejidad y poder, están las funciones **gets()** y **puts()**. Estos permiten leer y escribir cadenas de caracteres en la consola.

La función **gets()** lee una cadena de caracteres entradas y las escribe en la direccion apuntada por su argumento apuntador. El prototipo de **gets()** es:

char *gets(char *s);

La función gets retorna un puntero de **s**
la función **puts()** escribe el argumento de su cadena en pantalla seguido por una nueva linea. Su prototipo es:

int puts(char *s);

Reconoce los mismos códigos backslash como **printf()** como "\t"para tab. una llamada de **puts()** requiere mucho menos sobrecarga que la misma llamada de **printf()** porque **puts()** solo puede imprimir una cadena de caracteres- no puede imprimir números o hacer conversiones de formato. Por lo tanto la funcion **puts()** toma menos espacio y corre mas rapido que **printf()**. Por ejemplo:

puts("hello");

La funcion **puts()** usa el archivo de cabecera STDIO.H

14.5. Consola I/O formateada

El prototipo de **printf()** es:

int printf(char *control_string,...)

Notese los 3 puntos En el prototipo de **printf()**. cuando una función puede tomar un numero variable de argumentos, entonces su prototipo usa los tres puntos para especificar esto.
En la función **scanf()** las cadenas serán leídos en arreglos de caracteres, y el nombre de un arreglo sin algún indice, es la dirección del primer elemento del arreglo. Entonces para leer una cadena en la dirección del arreglo de la caracteres, podrías usar lo siguiente:

scanf(" %s", address);

En este caso **address** es ya un puntero, y no necesita ser precedido por el operador &.
Los datos de entrada deben ser separados por espacios, tabs o nuevas lineas. puntuación como comas, semi-columnas y similares, no cuentan como separadores. Esto significa que:

scanf(" %d %d", &r, &c);

Aceptará una entrada de 10 20, pero fallará con 10,20

La función **scanf()** incluye un a característica muy poderosa llamada *scanset*. un scanset define una lista de caracteres que serán alineados por **scanf()**. La función **scanf()** seguirá leyendo caracteres mientras estén en el scanset. Tan pronto como un carácter de entrada no se alinea con alguno del scanset, **scanf()** se mueve hacia el siguiente. Un scanset es se define poniendo una lista de los caracteres que quieras escanear dentro de paréntesis cuadrados. El inicio de los paréntesis cuadrados tiene como prefijo un signo de porcentaje, por ejemplo este scanset le dice a **scanf()** leer solo los dígitos desde 0 hasta 9:

%[1234567890] Esta declaración solo permitirá que esta cadena sea leída por la variable indicada

Puedes especificar un rango dentro de una scanset usando un guion medio. por ejemplo:

%[A-Z]

Esto le dice a **scanf()** que solo acepte caracteres desde A hasta Z. Ademas se puede especificar mas de un rango en el scanset. Por ejemplo:

...\learn c\c scanf_range.c

Algo importante de recordar es que scanset es case-sensitive, entonces si quieres poner mayúsculas y minúsculas individualmente.

14.6. El sistema ANSI I/O

El sistema ANSI I/O está compuesta de varias funciones interrelacionadas. Estas funciones requieren que el archivo de cabecera STDIO.H son incluidas por cualquier programa en donde son usadas.

14.6.1. El apuntador de archivo

Un común hilo que ata el sistema ANSI C I/O es el apuntador de archivo. Un apuntador e archivo es un puntero a información que define varias cosas sobre el archivo, incluyendo su nombre, estado y actual posición. En esencia, un archivo puntero identifica un archivo en disco duro y es usado por un stream asociado con el para la operación directa de funciones ANSI C I/O. un archivo puntero es un puntero de variable tipo **FILE**, que esta definido en STDIO.H. Para leer o escribir archivos, tu programa necesitará usar punteros de archivo. Para obtener variables puntero de archivo se utiliza una sentencia como la siguiente:

FILE *fp:

Function	**Operation**
fopen()	Opens a stream
fclose()	Closes a stream
putc()	Writes a character on a stream
getc()	Reads a character from a stream
fseek()	Seeks to the specified sbyte in a stream
fprintf()	Is to stream what **printf()** is to the console
fscanf()	Is to stream what **scanf()** is to the console
feof()	Returns true if the end of the file is reached
ferror()	Returns true if an error has ocurred
fread()	Reads a block of data from a stream
fwrite()	Writes a block of data from a stream
rewind()	Resets the file position locator to the begining of the file
remove()	Erates a file

14.6.2. Abriendo un archivo

La funcion **fopen()** sirve a dos propósitos. Primero abre un stream para usarlo y lo conecta a un archivo con ese stream. Segundo, retorna el archivo puntero asociado con ese archivo. Mas comunmenteel archivo es de disco. la funcion **fopen()** tiene este prototipo:

FILE *fopen(char *nombre-archivo, char *modo);

Donde *modo* es una cadena conteniendo abierto el estado deseado. El nombre de archivo debe ser una cadena de caracteres que comprometa un nombre de archivo valido para el sistema operativo y debe incluir especificación de dirección

Mode	Meaning
"r"	Open a file for reading
"w"	Create a file for writing
"a"	Append to a file
"rb"	Open a binary file for reading
"wb"	Create a binary file for writing
"ab"	Append to a binary file
"r+"	Open a file for write/read
"w+"	Create a file for read/write
"a+"	Append or create a file fore read/write
"r+b"	Open a binary file for read/write
"w+b"	Create a binary file for read/write
"a+b"	Append or create a binary file fore read/write
"rt"	Open a text file for reading
"wt"	Create a text file for writing
"at"	Append to a text file
"r+t"	Open a text file for read/write
"w+t"	Create a text file for read/write
"a+t"	Append or create a text file for read/write

Si quisieras abrir un archivo para escribir con nombre **test** entonces escribirias:

```
FILE *fp;

fp = fopen("test", "w");
```

Sin embargo, usualmente lo varas escrito así:

```
FILE *fp

if ((fp = fopen("test", "w")) == NULL)
{
puts("cannot open file\n");
exit(1);
}
```

La macro **NULL** está definida en STDIO.H. Este método detecta cualquier error abriendo un archivo, por ejemplo de protegido de escritura o disco lleno antes de intentar escribir en el. Un NULL es usado porque ningún archivo puntero tendría ese valor. También está la función **exit()**. Una llamada a **exit()** causa la inmediata terminación del programa, no importa de donde es llamada la función **exit()**, tiene un prototipo (encontrada en STDLIB.H):

```
void exit(int val);
```

Este valor es retornado al sistema operativo.

Si usas **open()** para abrir un archivo para escribir, entonces cualquier archivo preexistente por ese nombre será borrado y comenzará un nuevo archivo. Si no existe un archivo por ese nombre, entonces uno será creado. Si quieres añadir a el final de un archivo, entonces debes usar el modo "a". Abrir un archivo para leer operaciones, requiere que el archivo exista. Si no lo hace, un error será retornado. Finalmente, si un archivo es abierto para leer/escribir operaciones, este no será borrado si existe; sin embargo, si no existe se creará uno.

14.6.3. Escribir un caracter

la función **putc()** es usada para escribir caracteres a un stream previamente abierta para escribir por la función **fopen()** la función es declarada

int putc(int *ch*, FILE **fp*);

Donde *fp* es el archivo puntero retornado por **fopen()** y *ch* es el carácter de salida. El archivo puntero le dice a **putc()**que archivo de disco escribir en.Por razones históricas, *ch* es formalmente llamado un **int** pero solo el byte de bajo orden es usado.

Si una operación **putc()** es un éxito entonces va a retornar el carácter escrito. En un fallo, un **EOF** es retornado. **EOF** es un macro definido en STDIO.H que significa "End-Of-File"

14.6.4. Leyendo un caracter

la función **getc()** es usado para leer caracteres de un stream abierto en modo de lectura por **fopen()**. la función es declarada como:

int get(FILE **fp*);

Donde *fp* es un archivo puntero de tipo **FILE** retornado por **fopen()**. Por razones históricas, **getc()** retorna un entero, pero el byte de alto orden es cero.

la función **getc()** retornará una marca **EOF** cuando el final del archivo ha sido alcanzado o un error ha ocurrido. Por lo tanto para leer un archivo de texto hasta que la marca End-Of-File es leída, puedes usar el siguiente codigo:

```
ch = getc(fp);

while(ch != EOF) {
ch = getc(fp);
}
```

14.6.5. Usando feof()

Un archivo de sistema ANSI puede también ser operado en dato binario. Donde una archivo es abierto por entrada binaria, es posible que un valor entero igual a la marca **EOF** puede ser leído. Esto causaría la previa rutina de indicar una condición EOF aun cuando el final físico de el archivo no ha sido alcanzado. Para resolver este problema, ANSI C incluye la función **feof()**, donde es usado para determinar el final de el archivo cuando se leen datos binarios. La función **feof()** tiene este prototipo:

int feof(FILE **fp*);

Este prototipo está en STDIO.H. Retorna true si el final del archivo ha sido alcanzado; de otra manera, cero es retornado. Por lo tanto, la siguiente rutina lee un archivo binario hasta el final de que el final del archivo es encontrado.

```
while(!feof(fp)) ch = getc(fp);
```

por supuesto, este mismo método puede ser aplicado para archivos de texto así como archivos binarios.

14.6.6. Cerrando un archivo

La función **fclose()** es usado para cerrar un stream que ha sido abierto por una llamada a **fopen()**. Escribe cualquier dato aun en el búfer del disco (En informática, un búfer (del inglés, buffer) es un espacio de memoria, en el que se almacenan datos de manera temporal, normalmente

para un único uso su principal uso es para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede sin datos durante una transferencia de datos I/O irregular o por la velocidad del proceso.) a el archivo y hace un cierre formal a nivel de sistema operativo en el archivo. fallar en el cierre de un stream invita a todo tipo de problemas incluyendo perdida de datos, destruir archivos y posibles intermitentes errores en tu programa. **fclose()** también cierra el archivo de bloque de control asociado con el stream y lo hace posible para rehúso. Posiblemente sea necesario cerrar on archivo antes de abrir otro.

La funcion **fclose()** es declarada como:

int fclose(FILE *fp);

Donde *fp* es el archivo puntero retornado por la llamada a **fopen()**. Un retorno de valor cero significa un exito cerrando operacion; cualquier otro valor indica error. Puedes usar la funcion estandar **ferror()** para determinar y reportar cualquier problema. Generalmente, el unico momento **fclose()** fallará es cuando un diskette (o su analogo moderno) ha sido prematuramente removido del drive o si no queda espacio en el diskette (o su analogo moderno).

14.6.7. ferror() y rewind()

la funcion **ferror()** es usada para determinar si una operación de archivo ha producido un error. Si un archivo es abierto en modo texto y un erroe en lectura o escritura ocurre, se retorna **EOF**. se usa **ferror()** para determinar que evento pasó. La funcion **ferror()** tiene el prototipo:

int ferror(FILE *fp);

Donde *fp* es un archivo puntero valido. Retorna true si un error ha ocurrido durante la ultima operacion de archivo; retorn falso de otra manera. Porque cada operacion de archivo hace la condicion error, **ferror()** debería ser llamada inmediatamente despues de cada operacion de archivo; de otra manera, un error puede perderse. EL prototipo para **ferror()** está en STDIO.H.

la funcion **rewind()** reseteará el localizador de posicion de archivo a el principio de el archivo especificado como su argumento. Su prototipo es:

void rewind(FILE *fp);

Donde *fp* es un archivo punero valido. El prototipo para **rewind()** está en STDIO.H'

14.6.8. Usando fopen(), getc(), putc() y fclose()

las funciones **fopen()**, **getc()**, **putc()** y **fclose()** comprometen el minimo set de rutinas de archivo. Un simple ejemplo de usar **fopen()**, **putc()** y **fclose()** es el programa **ktod** abajo. Lee lacarteres desde el teclado y los escribe en un archivo de disco hasta que el signo de dolar es escito. EL archivo es especificado por el comando line. Por ejemplo, si llamas este programa **ktod**, entonces escribir **ktod test** te permitirá pasar lineas de texto hacia el archivo llamado test

...\learn c\c ktod.c

El programa complementar **dtos**, leerá cualquier archivo ASCII e imprime los contenidos en la pantalla.

...\learn c\c dtos.c

El siguiente programa copiará un archivo de cualquier tipo. Notese que los archivos son abiertos en modo binario y que **feof()** es usado para checar por el final del archivo.

...\learn c\c feof.c

14.6.9. Using get() and putw()

En adición a **getc()** y **putc()**, están las funciones **putw()** y **getw()** estas funciones no están definidas por el ANSI C estardar. Estas son usadas para leer y escribir enteros de y desde un archivo de disco. Estas funciones trabajan exactamente como **putc()** y **getc()** excepto que en lugar de leer o escribir un solo carácter, estas leen o escriben un entero. Estas tienen el siguiente prototipo:

int putw (int i, FILE **fp*);

int getw(FILE **fp*);

El siguiente fragmento de codigo escribirá un entero a el archivo de disco puntero por *fp*:

putw(10, fp);

los prototipos por **getw()** y **putw** estan en STDIO.H

14.6.10. fgets() y fputs()

El sistema ANSI de C I/O incluye dos funciones que pueden leer y escribir cadenas desde streams: **fgets()** y **fputs()**. Sus prototipos se muestran aqui.

char *fputs(char *str, FILE **fp*);
char *fgets(char *str, int *length*, FILE **fp*);

La función **fputs()** trabaja como **puts()** excepto que escribe la cadena el el stream especificado. La funcion **fgets()** lee una dadena desde el stream especificado hasta que un caracter nueva linea es leido o hasta que carecteres *length-1* han sido leidos. Si una nueva liea es leida, se volverá parte de la cadena. La cadena resultante será terminacion null. Los prototipos para **fgets()** y **fputs()** están en STDIO.H.

14.6.11. fread() y fwrite()

El sistema ANSI I/O provee dos funciones, llamadas **fread()** y **fwrite()**, que permiten la lectura y escritura de bloques de daos. Sus prototipos se miestran aquí:

unsigned fread(void ***buffer*, int *num_bytes*, int *count*, FILE **fp*);

unsigned fwrite(void ***fbuffer*, int *num_bytes*, int *cout*, FILE **fp*);

En el caso de **fread()**, *buffer* es un puntero a una región de memoria que va a recibir los datos desde el archivo. Para **fwrite()**, *buffer* es un puntero a la información que será escrito en el archivo. El numero de bytes a ser leídos o escritos es especificado por *num_bytes*. El argumento *count* determina cuantos objetos pueden ser leidos o escritos. Finalmente, *fp* es un archivo puntero a un stream abierto previamente. Ambas funciones tienen sus protipos en STDIO.H.

La funcion **fread()** retorna el numero de objetos leidos, que pueden ser menos que *count* si el final de el archivo es alcanzado o un error ocurrio. La funcion **fwrite()** retorna el numero de objetos escritos. Este valor será igual a *count* a no ser que un error ocurra.

Mientras el archivo sea abierto como dato binario, **fread()** y **fwrite()** pueden leer y escribir cualquier tipo de informacion. Por ejemplo este programa escribe un **float** en el disco:

...\learn c\c fwrite_fread.c

Como este programa ilustra, el buffer puede ser y seguido es, un a simple variable. Este programa tambien introduce otro operador de C: **sizeof**. El operador de tiempo de compilación **sizeof** retorna el tamaño en bytes de la variable o el tipo de dato que precede.

una de las mas utiles aplicaciones de **fread()** y **fwrite()** involucra leer y escribir arreglos (o como verás despues, estructuras). Por ejemplo. este fragmento escribe los contenidos de un arreglo de punto flotante **sample** a el archivo **sample** usando un solo **fwrite()**:

...\learn c\c sample_read.c

14.6.12. fseek() y acceso aleatorio I/O

Puedes hacer lectura y escritura aleatoria de operaciones usando el sistema ANSI C I/O con la ayuda de **fseek()**, que establece el localizador de posición del archivo. Su prototipo es:

int fseek(FILE *_fp_, long _numbytes_, int _origin_);

Aquí *fp* es un archivo puntero retornado por una llamada a **fopen()**; *numbytes* un entero long es el numero de bytes desde *origin* para hacer la posicion actual; *origin* es uyno de los siguientes macros definidos en STDIO.H:

Origin	**Macro Name**	**Actual Value**
Begining of file	SEEK_SET	0
Current position	SEEK_CUR	1
End of file	SEEK_END	2

Por lo tanto para buscar *numbytes* desde el inicio del archivo, *origin* debería ser **SEEK_SET**, para buscar desde la posicion actual debería ser **SEEK_CUR**, para buscar desde el final debería ser **SEEK_END**.

El siguiente fragmento lee el byte 235 en un archivo que es llamado **test**.

```
...
FILE *fp;
char ch;

if((fp = fopen("test", "rb")) == NULL) {
printf(Cannot open file\n);
exit(1);
}

fseek(fp, 234, 0);
ch = getch(fp);
...
```

Un valor 0 de retorno significa que **fseek()** tuvo éxito. Un valor diferente de 0 indica fallo.

Un mas interesante ejemplo es el programa **DUMP**, que usa **fseek()** para dejarte examinar los contenidos en ASCII y hexadecimal de cualquier archivo elijas. Puedes mirar el archivo en sectores de 128bytes en cualquier dirección. La salida es similar en estilo a el formato usado por DEBUG cuando dado el comando "D"(dump memory):

...\learn c\c dump.c

14.6.13. Los streams estandar

Cuando un programa empieza su ejecución, 5 streams son abiertos automaticamente. Los primeros 3 son entrada estandar(stdin), salida estandar (stdout) y error estandar (stderr). Normalmente, estos refieren a la consola, pero pueden ser redireccionados por el sistema operativo para hacer otro dispositivo stream. Porque estos son archivos punteros pueden ser usados por el sistema ANSI I/O

para realizar operaciones I/O en la consola. Por ejemplo, **putchar()** puede ser definido como:

```
putchar(char c)
{
putc(c, stdout);
}
```

Puedes usar **stdin**, **stdout** y **stderr** como archivos punteros en cualquier funcion que usa una variable de tipo **FILE ***.

Las funciones I/O de consola **getchar()**, **putchar()**, **printf()** y **scanf()** de hecho realizan sus operaciones I/O usando **stdin** y **stdout**. Desde que DOS permite redireccionar I/O usando los operadores de linea de comando > y <, estas funciones pueden tambien ser leidas y escritas en archivos de disco. Por ejemplo:

```
...\learn c\c iotest.c
```

Como puedes ver, nada se despliega en la pantalla. Sin embargo si enlistas los contenidos **OUT** verás que el mensaje ha sido escrito en el.

Debes mantener en mente que **stdin**, **stdout** y **stderr** no son variables sino constantes. Ademas, tambien así como son creados al inicio de tu programa automaticamente, tambien son cerrados automaticamente al terminar el mismo; no debes intentar cerrarlos tu mismo.

14.6.14. fprintf() y fscanf()

En adicion a las funciones basicas I/O discutidas, el sistema ANSI C I/O incluye **fprintf()** y **fscanf()**. Estas funciones funcionan exactamente como **printf()** y **scanf()**, excepto que estas operan con archivos de disco. Los prototipos de **fprintf()** y **fscanf()** son:

```
int fprintf(FILE *fp, char *control_string,...);
int fscanf(FILE *fp, char *control_string,...);
```

Donde **fp* es un archivo punter retornado por una llamada de **fopen()**. Excepto por dirigir su salida a el archivo definido por **fp*, estas operan exactamente como **fprintf()** y **fscanf()**, respectivamente.

Para ilustrar lo utiles que estas funciones pueden ser, el siguiente programa mantiene un simple directorio telefonico en un archivo de disco. Podrias ingresar numeros o puedes buscar un numero dado un nombre:

```
...\learn c\c fprintf_fscanf.c
```

Aunque **fprintf()** y **fscanf()** seguido son la forma mas sencilla de escribir y leer datos a archivos de disco, estos no son siempre los mas eficientes. Por que el sistema formateado ASCII está siendo escrito directamente como aparecería en pantalla, en lugar de en binario. Entonces si la velocidad o el tamaño del archivo es una preocupacion, deberias probablemante usar **fread()** y **fwrite()**.

14.6.15. Borrando archivos

la función **remove()** borra el archivo especificado. su prototipo es

```
int remove(char *filename);
```

retorna cero si es un exito, no-cero si falla

14.7. Las Rutinas de archivo tipo UNIX

Ya que C fue diseñado inicialmente bajo el sistema operativo UNIX, un segundo archivo de disco de sistema I/O fue creado. ysa funciones que son separadas de las funciones del sistema de

archivos ANSI

15. Tipos de dato avanzados

15.1. Modificadores de acceso

C tiene 2 tipos de modificadores de acceso que son usadas para controlar las formas en que las variables pueden ser accesadas o modificadas. Estos modificadores son llamados **const volatile**. Tambien son comunmente referidas como *cualificadores de tipo*

15.1.1. const

Variables declarados con el modificador **const** no podrian ser cambiadas durande la ejecucion de tu programa. Podrias darles un valor inicial, por ejemplo:

const float version = 3.20;

Crea una variable **float** llamada **version** que no podria ser modificada por tu programa. Puede sin embargo ser usada en otro tipo de expresiones.

Las variables de tipo **const** tienen un uso muy importante, estas pueden proteger los argumentos a una funcion de ser modificado por esa funcion. Esto es, cuando un puntero es pasado a una funcion, esto es posible para esa funcion modificar la variable apuntada por el puntero. Sin embargo si el puntero es especificado como **const** en la declaracon de parametros su valor no podra ser modificado, por ejemplo:

...\learn c\c const.c

Si por alguna razon intentas modificar el argumento, no compilara correctamente e imprimira el siguiente error

Cannot modify a const object in the function *name*

El segundo uso para **const** es para verificar que de hecho tu programa no modifica una variable. Recuerda que una variable puede ser modificada por algo fuera de tu programa, sin embargo de esta forma aseguras que ningun cambio en la variable es por tu programa, sino por eventos externos.

15.1.2. volatile

El modificador **volatile** es usado para decirle al compilador que el valor de una variable puede cambiar en formas que no estan especificadas por el programa. Por ejemplo, la direccion de una variable global puede ser pasada a la rutina de el sistema operativo y usada para mantener el tiempo real del sistema. En esta situacion los contenidos de la variable seran alterados sin algun asignamiento explicito en el programa. Ahora considera el siguiente fragmento de codigo:

int clock, timer;
.
.
.
timer = clock;
/*do Something*/
printf("Elapsed time is %d\n", clock-timer);

Porque **clock** no es alterada por el programa y no esta declarada como **volatile** Puede ser optimizado de tal forma en que el valor de **clock** no es reexaminado por **printf()**. Sin embargo si declaras **clock** como:

volatile int clock;

Entonces no ocurrira esa optimizacion y el valor de **clock** sera reexaminado cada vez que sea referenciado

Aunque pueda parecer extraño, es posible utilizar **const** y **volatile** juntas. Por ejemplo, si 0x30 es asumido ser el valor de un puerto que es cambiado solo por condiciones externas, entonces la siguiente declaracion es precisamente lo que quieres para prevenir cualquier posible efecto secundario accidental:

const volatiole unsigned har *port=0x30;

15.2. Especificadores de guardado de clase

Hay 4 especificadores de guardado de clase soportados por C:

- auto
- extern
- static
- register

Estos son usados para decirle al compilador como la variable que sigue deberia ser guardada. El especificador de guardado precede al resto de las declaraciones de variable. Su forma general es:

storage-class-soecufuer type-specifier variable-list;

15.2.1. auto

El especificador **auto** es usado para declarar variables locales. Sin embargo este es raramente usado porque las variables locales son **auto** por default. Es extremadamente raro ver esta palabra clave en un programa

15.2.2. extern

Todos los programas con los que has estado trabajando hasta el momento han sido pequeños. sin embargo en verdadera tareas de programación, los programas tienen a ser mucho mas largos. hasta el punto en que sin importar lo veloz que sea tu compilador, el tiempo de compilación crece tanto que se vuelve molesto. Cuando esto sucede deber romper tu programa en 2 o mas archivos separados. De esta manera pequeños cambios en un archivo no requiere que todo el programa sea recompilado, lo que significa salvar mucho tiempo en largos proyectos.

C contiene la palabra clave *extern*, que ayuda a soportar el acercamiento de múltiples archivos.

Ya que C permite separadamente compilar módulos de un programa largo para ser unido junto para acelerar la compilación y la ayuda en el manejo de largos proyectos, debería haber alguna manera de decirle a todos los archivos sobre las variables requeridas para el programa. Tu programa puede solo tener una copa de cada variable global. Si tratas de declarar dos variables globales con el mismo nombre en el mismo archivo, el compilador simplemente elije una y la usa (En C++ sin embargo es un error el declarar 2 variables con el mismo nombre). Si tu tratas de declaras las variables globles necesitadas por tu programa en cada archivo de un programa multi-archivo, estarás en problemas. Aunque el compilador no manda un mensaje de error en tiempo de compilación, estas tratando de crear 2 o mas copias de cada variable. Esto será encontrado por el enlazador cuando intente unir tus módulos juntos. El enlazador mandará un mensaje de error porque no sabrá que variable usar. La solución es declarar todas tus globales en un archivo y usar **extern** en los demás archivos.

15.2.3. Variables static

Variables de tipo **static** son variables permanentes en ya sea su propia funcion o el archivo, estas difieren de las varibles globlesporque no son conocidas fuera de su funcion o archivo, pero si mantienen sus valores entre llamadas. esta caracteristica puede hacerlas bastante utiles cuando estribes funciones generanlizadas y librerias de funciones, que pueden ser utilizadas por otros programadores.

15.2.4. Variables locales static

Cuando el modificador **static** es usado a una variable local, hace que el compilador cree un almacenamiento permanente en basicamente la misma forma en que hace con las variables globales. La principal diferencia entre una variable **static** y una variable global es que la variable **static** se mantiene solo en el bloque en que es declarada. En otras palabras, una variable local **static** es una variable local que mantiene su valor entre llamadas. Por ejemplo:

...\learn c\c static.c

15.2.5. Variables globales static

cuando el especificador **static** es aplicado a una variable global, le instruye al compilador crear una variable que es conocida solo por el archivo en el que se declara. Esto significa que aunque la variable sea local, otras rutinas en otros archivos, no podrian tener conocimiento de el o alterar su valor directamente.

Las variables **static** te permiten esconder porciones de tu codigo de otras porciones. Esta puede ser una tremenda ventaja cuando estas escribiendo para manejar un muy largo y complejo programa. El especificador de almacenamiento **static** te permite crear funciones muy generales que puedes guardar en librerias para su posterior uso.

15.2.6. Variables register

Otro importante tipo de modificador encontrado en C es llamado **register** y es tradicionalmente aplicado a variables de tipo **int** y **char**. El modificador **register** le pide al compilador acceder a esa variable lo más rápido posible, esto para los tipos de dato **int** y **char** significa que son guardadas en el cache del cpu

El especificador register solo es aplicable para variables locales y sus parametros, variables globales **register** no son validas es importante saber que **register** no es un comando, es una peticion, pues exsiste un limitado numero de locaciones de acceso rapido. cuando estos espacios se terminen, las demas variables declaradas con **register** se trataran como cualquier otra variable. Ejemplo:

...\learn c\c register.c

15.3. Punteros de funciones

Una particularmente confusa y aun así poderosa cualidad de C es el **archivo puntero**. Una funcion puntero es, de cierta forma, un nuevo tipo de dato. Aun cuando una funcion, no es una variable, aun tiene una locacion fisica en memoria que puede ser asignada a un puntero. La direccion asignada al puntero es el punto de entrada de la funcion. Este puntero puede despues ser usado en lugar del nombre de la funcion. Tambien permite a la funcion ser pasada como argumento de la funcion.

Veamos como se llaman y se compilan las funciones en C. Primero cada funcion es compilada, codigo fuente es transformado a un objeto y un punto de entrada es establecido. Cuando una llamada se hace a la funcion mientras tu program corre, una "llamada.[en] lenguaje maquina es hecha a este punto de entrada. Por lo tanto un puntero a una funcion de hecho contiene la direccion de memoria a el punto de entrada de la funcion.

La direccion de una funcion es obtenida usando el nombre de la funcion sin algun parentesis o argumentos. Por ejemplo:

...\learn c\c function_pointer.c

...\learn c\c function_pointer2.c

15.4. Asignación dinámica de memoria

Antes de abandonar el tema de tipos de dato avanzados, es necesario discutir sobre la asignación dinámica de memoria de C, que nos permite la creación de variables durante la ejecución del programa.

Existen 2 maneras principales en que un programa de C puede guardar información en la memoria principal de la computadora. La primera usa variables globales y locales que estan definidas por el lenguaje C, en ambos casos se requiere que el programador sepa preferentemente la cantidad de espacio de almacenamiento necesario para cualquier situación. La segunda manera es atravez de el uso del sistema de asigamiento dinamico de memoria de C.

15.5. Asignación y liberación de memoria

El centro del sistema de asignación dinámica de memoria de C son las funciones **malloc()** y **free()**. Estas trabajan juntas usando la region de memoria libre para establecer y mantener una lista de almacenaje disponible. La funcion **malloc()** asigna la memoria y **free()** la libera. La funcion **malloc()** tiene su prototipo:

void *malloc(unsigned *number_of_bytes*);

Retorna un puntero de tipo void, lo que significa que puedes asignarlo a cualquier tipo de puntero. Puedes usar **sizeof()** para determinar exacto numero de bytes necesarios para cada tipo de dato. De esta manera puedes hacer tus programas portables a una variedad de sistemas.

La función **free()** es el opuesto de **malloc()**. una vez la memoria ha sifo liberada puede ser usada por una subsecuente llamada a **malloc()**. La función **free()** tiene el siguiente prototipo:

void free(void *p);
ejemplo:

...\learn c\c free.c

La locacion dinamica de memoria es muy buena cuando no sabes por adelantado, con cuantos objetos de dato estas tratando. Aunque ejemplos importantes de la locacion dinamica suelen ser largos y complejos, el siguiente ejemplo te dará el sabor de su uso:

...\learn c\c free2.c

16. Tipos definidos por el usuario

C te permite crear 5 diferentes tipos de datos customizados. El primero es el *structure*, que es un agrupador de variables bajo un mismo nombre. El segundo tipo es el *bit-field*, que es un tipo de estructura que permite acceso sencillo a los bits en una palabra. El tercero es el *union*, que permite a la misma pieza de memoria ser definida como 2 o mas tipos de variables. El cuarto tipo es el *enumeration*, que es una lista de simbolos. El quinto tipo de archivo es creado a travez del uso de **typedef** y crea un nuevo nombre para un tipo existente.

16.1. Structures

Una estructura es una colección de variables que se referencian bajo un nombre, brindando una forma conveniente de mantener la información relacionada junta. Una *Estructura de declaración*, forma una plantilla que puede ser usada para crear estructuras de variables. Las variables que comprenden la estructura son llamados elementos de estructura o miembros de estructura. La palabra clave **struct** le dice al compilador que una estructura plantilla ha sido creada, ej:

```
struct addr{
char name[30];
char street[40];
char city[20];
char state[30];
unsigned long int zip;
};
```

La declaracion es terminada por un punto y coma. Esto es porque la declaración de una estructura es un estatuto, donde *addr* es el especificador de tipo e identifica esa particular estructura de datos.

Hasta este punto no se han declarado variables. Solo se ha definido la forma de la estructura de datos, para declarar variables se haría por ejemplo:

```
struct addr{
char name[30];
char street[40];
char city[20];
char state[30];
unsigned long int zip;
}addr_info, binfo, cinfo;
```

Si solo necesitas una variable estructura, entonces el identificador de tipo no es necesario, quedando en este caso:

```
struct{
char name[30];
char street[40];
char city[20];
char state[30];
unsigned long int zip;
}addr_info;
```

16.1.1. Referenciando miembros de una estructura

Los miembros individuales de una estructura son referenciadas a travez de el uso de el operador de seleccion de miembro: el punto (tambien llamado *Operador punto*). Por ejemplo:

```
addr_info.zip = 12345;
```

Todos los elementos de estructuras se accesan de la misma manera, Teniendo la forma general de:

structure_varname.element_name

16.2. Arreglos de estructuras

El uso mas comun de estructuras es en arreglos de estructuras. Para declarar un arreglo de estructuras, primero debes definir una estructura y despues declarar una variable arreglo de ese tipo

...\learn c\c mailing_list.c

16.3. Assignando estructuras

Si 2 estructuras son de un mismo tipo, puedes entonces asignar la una a la otra. En este caso, todos los elementos de la estructura en el lado izquierdo de el asignamiento, recibirá los valores de sus correspondientes elementos de la estructura ej la derecha. por ejemplo:

...\learn c\c assigning_structures.c

Recuerda, no puedes asignar una estructura a otra si son de diferente tipo, incluso si comparten algunos elementos

16.4. Pasar elementos de estructuras a funciones

Hasta ahora, todas las estructuras y arreglos de estructuras se ha asumido que son globales o definidas en la funcion que las usa. En esta seccion se tomará en consideracion especial aa pasar estructuras y sus elemetos a funciones.

16.4.1. Pasando elementos de estructura a funciones

Pasar un elemento de una estructura, es solamente pasar una variable, a no ser que sea un elemento complejo, como un arreglo de caracteres; sin embargo, si deseas pasar la direccion de un elemento de la estructura, tienes ue poner el operador & antes del nombre de la variable de estructura, no del elemento

16.4.2. Pasar estructuras enteras a funciones

Cuando una estructura es pasada como argumento de una funcion, toda la estructura es pasada usando la llamada estandar por valor. Esto por su puesto significa que cualquier cambio hecho a los conenidos de la estructura dentro de la funcion a la cual es pasada no afecta la estructura usada como un argumento por ejemplo:

...\learn c\c struct_param.c

16.5. Punteros a estructuras

C permite punteros a estructuras de la misma forma que permite punteros a cualquier otro tipo de variable. Sin embargo, existen cientos aspectos especiales de los que debes estar al tanto.

16.5.1. Declarando un puntero de estructura

Los punteros de estructura, son declarados colocando ***** frente al nombre de la variable de estructura por ejemplo:

struct addr *addr_pointer;

16.5.2. Usando punteros de estructura

Existen varios usos para los punteros de estructura, uno es el de archivar una llamada por referencia a una funcion. Otro es crear una lista linkeada y otras estructuras dinamicas de datos (*Turbo C/C++: The Complete Reference*)

Existe un gran contratiempo al pasar estructuras enteras a funciones y es el esfuerzo para mover toda la estructura de elementos. En simples estructuras, con algunos elementos, este esfuerzo no es demasiado importante, pero si es una estructura de varios elementos o si estos son arreglos, entonces este esfuerzo puede degradar el tiempo de ejecución a niveles inaceptables. La solución a este problema está en los punteros a estructuras. Ejemplo:

```
struct{
float balance;
char name;
}person;

struct bal *p = &person;
```

Para accesar a **balance** podrías escribir:

```
(*p).balance
```

Es extraño ver, si alguna vez lo haces, el encontrar que se utilice explicitamente el operador * para accesar a una estructura atravez de un puntero, es algo tan comun que de hecho existe un operador especial para esta tarea, es el -¿, al cual la mayoria de los programadores de C llaman el operador *flecha*. Por ejemplo, para escribir lo anterior se haría:

```
p->balance
```

Ejemplo:

```
...\learn c\c struct_pointer.c
```

Recuerda, debes usar el operador punto cuando trabajes con la estructura en si, cuando tengas un puntero a la estructura, utiliza el operador flecha, tambien recuerda que debes pasar la direccion de la estructura con el **&**.

Vamos a explorar algunas de las funciones de tiempo y fecha de C. Las funcionese que tratan con el sistema de tiempo y fecha, requieren el archivo de cabecera TIME.H para sus prototipos. Tambien se incluye en este archivo de cabecera 2 tipos definidos. El **time_t** es capaz de representar el sistema tiempo y fecha como un entero largo. este es referido como el *tiempo calendario*. La estructura de tipo **tm** mantiene la fecha y el tiempo partido en sus elementos. La estructura **tm** es definida de esta manera:

```
struct tm{
int tm_sec; // seconds, 0-59
int tm_min; // munutes, 0-59
int tm_hour; // hours, 0-23
int tm_mday; // day of the month 1-31
int tm_mon; // months since Jan, 0-11
int tm_year; // years form 1900
int tm_wday; // days since Sunday 0-6
int tm_yday; // days since Jan 1, 0-365
int tm_isdst; // daylight savings time indicator
}
```

El valor de **tm_isdst** será positivo si el tiempo de verano esta en efecto, 0 si está inactivo o un valor negativo si no se tiene informacion sobre ello. Esta forma de fecha y tiempo es llamada *broken-down time*.

la fundacion por el tiempo y la fecha de C es **time()** que tiene el prototipo:

time_t time(time_t *time);

La funcion **tiempo()** retorna el numero de segundos que han pasado desde enero 1 1970. Puede ser llamado con un puntero nulo o con un puntero a una variable de tiempo **time_t**

Para convertir de tiempo de calendario a *broken-down time*, usa **localtime()**, que tiene este prototipo:

struct tm *localtime(time_t *time);

La estructura usada por **localtime()** es estaticamente alocada y es sobreescrita cada vez que la funcion es llamada. Si deseas guardar los contenidos de la estructura, es necesario copiarlo en otro lado.

Aunque tu programa puede utilizar la forma *broken-down* de el tiempo y fecha, la forma mas sencilla de generar una cadena de tiempo y fecha es por el uso de **asctime()** cuyo protortipo es:

char *asctime(struct tm *ptr);

La funcion **asctime()** retorna un puntero a una cadena de la forma:

day month date hours:minutes:seconds year\n\0

El puntero de estructura pasada a **asctime()** es la obtenida por **localtime()**

Como **localtime()** es estaticamente alocada y es sobreescrita cada vez que la funcion es llamada. Si deseas guardar los contenidos de la estructura, es necesario copiarlo en otro lado. Ejemplo:

...\learn c\c time.c

16.6. Arreglos y estructuras en estructuras

Miembros de estructuras pueden ser de cualquier tipo de dato de C, incluyendo arreglos y estructuras. Por ejemplo:

```
struct x {
int a [10][10];
float b;
} y;
```

para referenciar al entero 3,7 en **a** de la estructura **y** escribirias:

y.a[3][7];

Cuando una estructura es elemento de otra estructura, se llama estructura *anidada*. Por ejemplo:

```
struct emp {
struct addr address;
float wage;
} worker;
```

Para referenciar a un elemento de *address* en **emp** escribirias:

worker.address.*variable*;

16.7. Bit-fields

A diferencia muchos otros lenguajes, C tiene un metodo intrinseco para accesar a uno o mas bits en un byte o palabra. Esto pude ser util para un numero de razones:

- Si el almacenamiento es limitado, puedes guardar varias variables *booleanas* en un byte
- Cierta interfaz de un dispositivo transmite informacion codificada en bits dentro de un byte
- Cierta encriptacion de rutinas necesita accesar los bits en un byte

Un metodo que C usa para accesar bits es basado en la estructura llamada *bit-field*. un *bit-field* es solo un tipo de miembro de estructura que define en bits que tal largo cada elemento es. La forma general de la declaracion de un campo de bits es:

struct *struct-type-name*{
type name1 : length;
type name2 : length;
...
type nameN : length;
}

Un campo de bits debe ser declarado como **int**, **unsigned** or **signed**. Campos de bits de longitud 1 deben ser declarados como **unsigned** porque un solo bit no puede tener un signo.

No tienes que nombrar cada campo de bits. Esto hace sencillo alcanzar el bit que quieres, pasando los no usados, por ejemplo:

```
struct equip{
unsigned floppy_boot: 1;
unsigned has8087: 1;
unsigned mother_ram: 2;
unsigned video_mode: 2;
unsigned floppies: 2;
unsigned dma: 1;
unsigned ports: 3;
unsigned game_adapter: 1;
unsigned : 1;
unsigned num_printers: 2;
} eq;
```

Las variables bit-field tienen ciertas restricciones:

- No puedes tomar la direccion de una variable bit-field.
- Las variables bit-field no pueden ser arraigadas.
- No puedes sobrelapar fronteras de enteros.
- Si el campo correrá de izquierda a derecha o de derecha a izquierda variará en diferentes CPU; Esto implica que cualquier codigo que use campos de bits puede tener algounas dependencias a maquina.

Es valido mezclar estructuras normales con miembros de campos de bits. Por ejemplo:

```
struct emp{
struct addr address;
float pay;
unsigned lay_off: 1;
unsigned hourly: 1;
unsigned deductions: 3;
};
```

Esta estructura define el record de un empleado que usa solo un byte para mantener 3 piezas de informacion: el estado del empleado, si es salariado y el numero de deducciones. Si el uso de campos de bits, esta información habria tomado 3 bytes.

16.8. Uniones

En C, una **union** es una locacion de memoria que es usada por diferentes variables de diferentes tipos. La declaracion de una **union** es similar a la de una estructura, por ejemplo:

```
union u_type {
int i;
char ch;
};
```

Como en las estructuras, esta declaracion no declara alguna variable. Puedes declarar una variable ya sea colocando su nombre al final de la declaracion o usando una declaracion separada. Para declarar una **union** variable **cnvt** de tipo **u_type** usando la declaracion ya dada, se escribiria:

```
union u_type cnvt;
```

En **cnvt**, ambos enteros **i** y el caracter **ch** comparten la misma locacion de memoria. (por lo tanto, **i** ocupa 2 bytes y **ch** ocupa solo 1)

Cuando una variable **union** es declarada, el compilador automaticamente creará una variable lo suficientemente larga para mantener el tipo de variable mas largo en la **union**

Para accesar a un elemento de la **union** usas la misma sintaxis que se usa para estucturas:

- Operador punto
- Operador flecha

Las uniones son usadas frecuentemente cuando conversiones de tipo son necesarios porque te dejan tomar una region de memoria en mas de una manera. por ejemplo:

```
void putw(union pw word, FILE *fp){
putc(word− >ch[0], fp);
putc(word− >ch[1], fp);
}
```

Aunque llamado con un entero, **putw()** puede aun usar la funcion estandar **putc()** para escribir un entero a un archivo de disco

el siguiente programa combina uniones con campos de bits para desplegar el codigo ASCII, en binario, generado cuando presionas una tecla. La **union** permite a **getche()** asignar el valor de la tecla a un caracter variable mientras el campo de bits es usado para desplegar los bits individuales (Este programa es solo para ilustrar como uniones pueden ser usadas para buscar en la misma pieza de memoria de 2 formas diferentes. Hay muchas maneras mas eficientes de escribir la funcion

deconde() que alcanzan el mismo resultado).

...\learn c\c union.c

16.9. Enumeraciones

Una enumeracion es un conjunto de nombres de constantes enteras y especifica todos los valores legales que una variable de tipo puede tener. Son definidas mucho como estructuras con la palabra clave **enum** usada como señal de inicio de un tipo de enumeracion, la forma general es mostrada aquí:

enum *enum-type-name* { *enumeration list* } *variable-list*;

la lista de enumeracion es una lista de nombres separada por comas que representa los valores que una variable de el tipo enumeracion puede tener. Ambos, la lista de tipo enumeracion y la lista de variables es opcional. Como en estructuras, el nombre de enumeracion de tipo es usado para declarar variables de su tipo. Por ejemplo:

enum coin{ penny, nickel, dime, quarter, half_dollar, dollar};

enum coin money;

Dadas estas declaraciones, los siguientes tipo de sentencias son perfectamente validas:

money = dime;

if(money == quarter) printf("is a quarter\n");
La clave para entender las enumeraciones es que cada uno de lso simbolos, se establece para un valor entero, así pueden ser usadps en cualquier expresion entera. A no ser que sea iniciado de otra forma, el primer simbolo inicia en cero, el valor del segundo es 1 y así continúa.

Una común pero errónea asunción es que los simbolos pueden ser input o output directamente. Por ejemplo, este codigo no preformará como se desea:

money = dollar;

printf(" %s", money);

16.10. Usando sizeof para asegurar portabilidad

Has visto que estructuras, uniones y enumeraciones pueden ser usadoas para crear variables de tamaños variables, y que el propio temaño de estas variables puede cambiar de maquina a maquina. El operador unario **sizeof** es usado para computar el tamaño de cualquier variable o tipo y puede ayudar a eliminar dependencia de maquina en el codigo de tu programa.
El operador **sizeof** toma las dos formas mostradas aquí:

sizeof *var-name*

sizeof (*type-name*)

16.11. typedef

C te permite explicitamente fedinir unevos tipo de datp usando la palabra clave **typedef**. No estás de hecho creando nuevos tipos de dato, sino definiendo un nuevo nombre para un tipo de dato ya existente. Este proceso puede ayudar a hacer programas dependentes de maquina mas portables; solo hace falta cambiar la sentencia de **typedef**. Tambien puede ayudar a documentar

tu codigo permitiendo nombres descriptivos para los tipos de dato estandar, la forma general de **typedef** es:

typedef *type name*;
Puedes usar typedef para definir tipos complejos de variables, por ejemplo:

```
typedef struct client_type{
float due;
int over_due;
char name[40];
} client;

client clist[NUM_CLIENTS];
```

En este caso, **client** no es una variable de tipo client_type, sino otro nombre para **struct client_type**.

Usando typedef puede ayudar a hacer tu codigo mas facil de leer y mas facil de portar a otra computadora. Pero recuerda que no estas nuevos tipos de dato.

17. Operadores avanzados

A diferencia de otros lenguajes, C contiene muchos tipos de operadores especiales que en gran medida aumenten su poder y flexibilidad -especialmente para programacion a nivel de sistema.

17.1. Operadores bitwise

A diferencia de muchos otros lenguajes, C soporta un completo arsenal de operadores bitwise. Como C fue diseñado para tomar el lugar del lenguaje ensamblador para la mayoria de as tareas de programacion, era importante que tuviera la hablidad de soportar todas (o al menos muchas) operaciones que pueden hacerse en ensamblador. *operadores bitwise* se refieren a testing, setting or shifting del propio bit en un byte o palabra, que corresponde en C a los tipos de dato char, int o long. Los operadores bitwise no deberias ser usados con los tipos de dato float, double, long double, void o otros tipos mas complejos:

Operator	**Action**
&	AND
¦	OR
∧	XOR
∼	NOT
>>	Shift right
<<	Shift left

Las operaciones bitwise son muy seguido encontradoes en drivers, programas modernos, rutinas de archivos de disco, rutinas de impresion y porque similares, pues estas pueden ser usadas para enmascarar ciertos bits, como una paridad. (La paridad de bits es usada para confirmar que el resto de los bits en el byte estan sin cambios. Esto es usualmente los bit de alto orden en cada byte)

En terminos de sus mas comunes usos, puedes pensar en el bitwise AND como una forma de apagar bits. Esto es que cualquier bit que es 0 en cualquier operando, causara el correspondiente bit en la variable ser establecido como 0.

"A"& 127

parity bit	
↓	
11000001	**ch** containing an "A"with parity set
01111111	127 in binary
&_____	do bitwise AND
01000001	"A"without parity

En general, bitwise ANDs, ORs y XORs aplican sus operaciones directamente en cada bit en la variable y no son usualmente usadas en sentencias condicionales de la manera en que los operadores relacionales logicos son usados. Por ejemplo, si $x = 7 \rightarrow x\&\&8 = true$, but $x\&8 = false$.

Los operadores relacionales y logicos, siempre dan como resultado 0 o 1, mientras que sus similares bitwise, pueden producir cualquier valor arbitrario en acordanza con la operacion especifica.

El operador AND es tambien util cuando quieres revisar si un bit está encendido o apagado. Por ejemplo:

```
if(status & 8) printf("bit 4 is on");
```

Un interesante uso de este proceso es la funcion **disp_binary()** mostrada aqui:

```
void disp_binary(){
for(int t = 128; t>0; t=t/2)
if(i & t) printf("1 ");
else printf("0 ");
printf("\n");
}
```

La funcion disp_binary() trabaja testeando cada bit en el byte, usando bitwise AND, para determinar si este está encendido o apagado. Si está encendido, el digito 1 es impreso, de otra forma, se imprime 0. En el capitulo anterior, el campo de bits fue utilizado para imprimir un valor binario (...\learn c\c union.c). El codigo mostrado aqui, es una mejor alternativa pues es mas rapido y pequeño.

los operadores de shift, >> y <<, mueven todos los bits en un valor integral, a la izquirerda o derecha como se especifique. La forma general de shift a la derecha es:

value >> number of bit positions

y a la izquierda es:

value << number of bit positions

las operaciones shift pueden ser muy utiles decodificando entradas externas, como convertidores D\A y leyendo informacion de estado. Las operaciones bitwise shift pueden ser tambien usadas para realizar muy rapidas multiplicaciones y divisiones de enteros. Un shift a izquierda multiplicará por 2 y un shift a derecha, dividirá sobre 2. Si el tras la multiplicacion es mayor al rango de su tipo, este tendrá una perdida de informacion.

El siguiente programa graficamente muestra el efecto de los operadores shift:

18. Estructuras

Una estructura es la colección de uno o mas elementos denominados miembros; estos pueden ser de un tipo de dato diferente. para declarar una estructura debemos utilizar la palabra reservada **struct**. para poder definir una estructura debemos de ponerle un nombre. en este caso las estructuras son muy parecidas a los objetos.

Ejemplo:

```
...
struct perro
{
char nombre[30];
int edadmeses;
float peso;
}perro1={“gato”,10,3.50},
perro2={“roberto”,4,2.30};
int main()
{
printf(“El peso de %s es %.2f kg y tiene %d meses \n”,perro2.nombre,perro2.peso,perro2.edadmeses);
return 0;
}
```

18.1. Arreglo de estructuras

Ejemplo:

```
...
struct perro
{
char nombre[30];
int edadmeses;
float peso;
}perros[3];
int main()
{
for (int i = 0; i < 3; ++i)
{
printf(“ %iIngresa el nombre del perro\n”,i+1);
scanf(“ %s”,&perros[i].nombre);
printf(“ %iIngresa la edad del perro\n”,i+1);
scanf(“ %i”,&perros[i].edadmeses);
printf(“ %iIngresa el peso del perro\n”,i+1);
scanf(“ %f”,&perros[i].peso);
}
for (int i = 0; i < 3; ++i)
{
printf(“ %i El nombre del perro es: %s\n”,i+1,perros[i].nombre);
printf(“ %i La edad del perro es: %i\n”,i+1,perros[i].edadmeses);
printf(“ %i El peso del perro es: %.2f\n”,i+1,perros[i].peso);
}
return 0;
}
```

18.1.1. Operador punto

Cada vez que declaramos un arreglo de estructuras para acceder a una variable utilizamos el punto.

18.2. Estructuras anidadas

Las estructuras anidadas siempre se encuentran dentro de otra estructura.

Ejemplo:

```
...
#define length 2
struct owner
{
char nombre[20];
char direccion[30];
};
struct dog
{
char nombre[20];
int edadmeses;
struct owner ownerDog;
}dogs[length];
int main()
{
for (int i = 0; i < length; ++i)
{
printf("Nombre del perro\n");
scanf("%s",dogs[i].nombre);
printf("Edad del perro en meses\n");
scanf("%i",&dogs[i].edadmeses);
printf("Nombre del dueño\n");
scanf("%s",dogs[i].ownerDog.nombre);
printf("direccion\n");
scanf("%s",dogs[i].ownerDog.direccion);
printf("\n");
}
for (int i = 0; i < length; ++i)
{
printf("El nombre del perro es: %s\n",dogs[i].nombre);
printf("Edad en meses del perro: %i\n",dogs[i].edadmeses);
printf("Nombre del dueño %s\n",dogs[i].ownerDog.nombre);
printf("La direccion es: %s\n",dogs[i].ownerDog.direccion);
}
return 0;
}
```

19. Asignación dinámica de memoria II

La memoria es un espacio que reserva nuestra computadora para almacenar algun valor o dato.

Podemos encontrar:

- Memoria estática
- Memoria dinámica

La **memoria estática** es la que venimos utilizando donde no nos preocupamos por el uso excesivo de la memoria. El problema viene cuando no utilizamos toda la memoria que se le fue asignada; por lo tanto la desperdiciamos, no estamos optimizando nuestro programa para poder utilizar un mínimo de memoria, por lo que tenemos la opción de asignar la memoria dinámica-mente.

La **memoria dinámica** (**malloc**) es un tipo de memoria que se reserva en tiempo de ejecución, así que su tamaño puede variar al momento de ejecutarse. Es importante utilizarlo cuando no sabemos el numero de datos o elementos que va a contener nuestro programa. Para utilizar malloc es necesario llamar a la librería *stdlib.h*

Ejemplo 1:

```
...
int main()
{
int n=10;
char * p;
p = malloc(n*sizeof(char));
if (NULL == p)
{
printf("Error al asignar memoria\n");
}else{
printf("Se asignó memoria\n");
}
return 0;
}
```

Ejemplo 2:

```
#define length 2
int size;
struct dog
{
char name[20];
char *p_name;
}dogs[length];
int main()
{
for (int i = 0; i < length; ++i)
{
printf("Ingrese el nombre del perro\n");
scanf(" %s",dogs[i].name);
size = strlen(dogs[i].name);
printf(" %i\n",size);
dogs[i].p_name = malloc(size * sizeof(char));
if (NULL == dogs[i].p_name)
{
printf("Error al asignar memoria\n");
}else{
strcpy(dogs[i].p_name,dogs[i].name);
}
}
for (int i = 0; i < length; ++i)
{
printf("El nombre del perro es: %s\n",dogs[i].p_name);
}
```

return 0;
}

19.1. Liberar memoria dinámica

19.2. *'fgets' y 'free'*

free es lo que se utiliza para liberar la memoria el cual solo necesita de la variable o el apuntador que se va a liberar como parámetro. Como opción a la utilización de ***scanf*** podemos utilizar ***gets***, el cual solo necesita un parámetro que es la dirección de donde se van a archivar los datos; pero puede ser inseguro pues no limita la cantidad de datos que entran al escanear sino se le especifica al arreglo. Sin embargo para evitar este problema utilizamos ***fgets*** el cual necesita de 3 parámetros:

- El puntero o la variable donde se va a almacenar la información.
- El tamaño de lo que esperamos recibir.
- De donde se van a obtener esos datos.

19.2.1. Operador

Cuando hacemos referencia a un puntero utilizamos el operador $->$ haciendo una pequeña flechita. Y de esta forma podemos liberar la memoria de la variable a la que apunta la flecha, pero si tratamos de acceder a ella nos va a ocasionar un problema.

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define length 2
int size;
struct dog
{
char name[20];
char *p_name;
}dogs[length];
int main()
{
for (int i = 0; i < length; ++i)
{
printf("Ingrese el nombre del perro\n");
fgets(dogs[i].name,20,stdin);
size = strlen(dogs[i].name);
printf(" %i\n",size);
dogs[i].p_name = malloc(size * sizeof(char));
if (NULL == dogs[i].p_name)
{
printf("Error al asignar memoria\n");
}else{
strcpy(dogs[i].p_name,dogs[i].name);
free(dogs[i].name);
{
}
for (int i = 0; i < length; ++i)
{
printf("El nombre del perro es: %s\n",dogs[i].p_name);
```

```
}
return 0;
}
```

20. Listas enlazadas

Son una colección de estructuras autorreferenciadas llamadas ***nodos***. en las listas enlazadas podemos guardar y eliminar datos en tiempo de ejecución y no es necesario definir cuantos espacios va a tener nuestra lista.

En esta definición hablamos de estructuras autorreferenciadas. ¿Que es una estructura autorreferenciada? Una estructura autorreferenciada contiene un miembro apuntador el cual apunta a una estructura del mismo tipo.

Ahora ¿Que es un nodo en C? Un nodo en C es una estructura que se crea con memoria dinámica; también al momento en que creamos estructuras nos podemos encontrar con algo llamado ***typedef*** que se antepone a la estructura nos permite crear sinónimos para nuestras estructuras.

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct nodo
{
char *nombre;
struct nodo *sig;
}Libro;
Libro *listaLibro(Libro *Lista){
Lista = NULL;
return Lista;
}
Libro *agregarLibro(Libro *Lista, char *nombre){
Libro *nuevoLibro, *aux;
nuevoLibro = (Libro*)malloc(sizeof(Libro));
nuevoLibro− >nombre = nombre;
nuevoLibro− >sig = NULL;
if (Lista == NULL)
{
Lista = nuevoLibro;
}else{
aux = Lista;
while(aux− >sig !=NULL){
aux = aux− >sig;
}
aux− >sig = nuevoLibro;
}
return Lista;
}
int main()
{
Libro *Lista = listaLibro(Lista);
Lista = agregarLibro(Lista,“HTML5 Avanzado”);
Lista = agregarLibro(Lista,“CSS3 Avanzado”);
while(Lista != NULL){
printf(“ %s\n”,Lista− >nombre);
Lista = Lista− >sig;
```

```
}
return 0;
}
```