Notas de C

Logan Martinez

15 de agosto de 2020

Resumen
Pues estas son mis notas para aprender lenguaje de progamación 'C' desde "cero".

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

15. Operador SizeOf

C proporciona el operador unitario **sizeof** para determinar el tamaño en bytes de un arreglo o cualquier otro tipo de dato durante la compilación del programa.

15.0.1. *size_t*

Es un tipo definido por el estándar de **C** como un tipo entero y el valor que debemos de retornar no debe tener signo entonces van a ser puros valores positivos.

Ejemplo:

```
...
size_t getsize(float *ptr);
int main()
{
float array[20];
printf("El número de bytes en el arreglo es: %lu\n",sizeof(array));
printf("El número de bytes devueltos por getsize es: %lu\n",getsize(array));
return 0;
{
size_t getsize(float *ptr)
{
return sizeof(ptr);
}
```

16. 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;
}
```

16.1. 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;
}
```

16.1.1. Operador punto

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

16.2. 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;
}
```

17. Asignación dinámica de memoria

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;
}
```

17.1. Liberar memoria dinámica

17.2. *'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.

17.2.1. 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;
}
```

18. 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;
}
```