Notas de C

Logan Martinez

17 de febrero 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 direccion. por lo que indica la variable en la que se va aguardar esa informacon.

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 estandar.

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

5.2. *Define*

La directiva **#define** crea constantes simbolicas 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. Ciclos

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

7.1. Ciclo *for*

Ejemplo:

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

7.2. Ciclo *while*

Nos permite especificar una accion mientras la condicion sea verdadera.

Ejemplo:

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

7.3. Ciclo *Do while*

Ejemplo:

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

8. *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 nuesto programa.

8.1. Etiqueta case

Aqui se escribe el nombre del caso; Si es algun caracter debe ir entre comillas o si es un numero puede ser escrito sin comillas.

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;
}
```

9. 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 estatica, ya que etos no cambiaran su tamaño durante la ejecucion del programa.

9.1. Arreglo unidimencional

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");
```

9.2. Arreglo multidimencional

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]);
```

10. *Break* y *Continue*

Nos permiten manejar el flujo de nuestro programa como querramos.

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

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

10.1. *Break*

Ejemplo:

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

En este caso se detiene el programa.

10.2. *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.

11. Funciones

11.1. Funciones basicas

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;
}
```

11.2. 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;
}
```

11.3. Funciones recursivas

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. Apuntadores

12.1. Conceptos basicos

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 = &2;
```

Se imprime como:

```
...
printf(“ %i\n”,*apt);
```

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

```
...
printf(“ %p\n”,apt);
```

12.2. Llamadas por referencia

Existen 2 maneras de pasar argumentos a una funcion.

- 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 estan 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;
}
```

13. Operador SizeOf

C proporciona el orepador unario **sizeof** para determinar el tamaño en bytes de un arreglo o cualquiero otro tipo de dato durante la compilacion del programa.

13.0.1. *size_t*

Es un tipo definido por el estandar 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);
}
```

14. Estructuras

Una estructura es una colección de uno o mas elementos denominados miembros; estos miembros 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;
}
```

14.1. 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;
}
```

14.1.1. Operador punto

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

14.2. 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;
}
```

15. Asignación dinamica de memoria

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

Podemos encontrar:

- Memoria estatica
- Memoria dinamica

La **memoria estatica** 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 minimo de memoria para utilizar nuestro programa. por lo que tenemos la opcion de asignar la memoria dinamicamente.

La **memoria dinamica** (**malloc**) es un tipo de memoria que se reserva en tiempo de ejecución, asi que su tamaño puede variar al momento de ejecutarce. 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 libreria *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;
}
```

15.1. Liberar memoria dinamica

15.2. *'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 parametro. Como opcion a la utilización de ***scanf*** podemos utilizar ***gets***, el cual solo necesita un parametro que es la direccion de donde se van a archivar los datos; pero puede ser inseguro pues no limita la cantidad de datos que entran al escanear sinó se le especifica al arreglo. Sin embargo para evitar este problema utilizamos ***fgets*** el cual necesita de 3 parametros:

- 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.

15.2.1. 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 ocacionar 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;
}
```

16. 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 dinamica; tambien al momento en que creamos estructuras nos podemos encontrar con algo llamado ***typedef*** que se antepone a la estructura nos permite crear sinonimos 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;
}
```