PROGRAMACIÓN

Días: Jueves y Viernes

Profesor: Yanina Scudero

Alumno: Miguel Pinto

División: 1D

Tipo de datos

- int (%d o %i): Valores enteros, números naturales -> 4 bytes de memoria -> 32 bits
- float (%f): Valores enteros o decimales -> 4 bytes de memoria -> 32 bits
- char (%c): Un solo carácter de memoria (‘a’, ‘5’, ‘@’) -> 1 byte -> 8 bits
- double (%Lf): Float más preciso, con más decimales -> 8 bytes -> 64 bits
- void: Vacío.

NO DECLARAR EN LA MISMA LÍNEA

Secuencias de escape

- (\: Alt + 92)
- \n: Salto de línea
- \t: Tabulación
- \a: Alarma -> hace un sonido al mostrar el mensaje
- \b: Retroceso
- \v: Tabulación vertical
- \\: Contrabarra
- \f: Salto de página
- \’: Apóstrofe
- \”: Comillas dobles
- \0: Fin de una cadena de caracteres

Pedir datos

scanf (“mascara”, &variable);

El uso de & le dice dónde guarda el valor

Casteo de datos

Transformar un tipo de dato a otro tipo de dato, poner (float) o (int) atrás de un número para modificarlo

```
promedio = (float)suma / 5;
```

const: Para asegurarse de que la variable (ahora constante) no cambie.

DEBE DECLARARSE EN LA MISMA LÍNEA. Consume más recursos.

Buffer

Entre el disp. De entrada y la memoria, existe una intermedia llamada “buffer”, lo que hace es que, cada vez que se ingresa algo, antes de llegar a la memoria pasa por el buffer.

Windows: fflush: Vacía el buffer

Buffer de entrada: stdin

Exclusiva de Windows

Buffer de salida: stdout

Linux (y GDB): __fpurge(stdin)

Funciones

Mini programa que se puede invocar desde otra función.

AMBITO O SCOPE DE VARIABLES

Depende de donde se declare las variables cambia su **visibilidad**, es decir desde donde se puede leer y escribir:

Global

- Se **declara** fuera de alguna función y debajo **del include**
- **No se considera una buena práctica** ya que se necesita tener mucho conocimiento y experiencia para usarlas de forma eficiente.
- Puede ser tratada por todo el proyecto
- Se debe declarar **debajo del include**

Locales

- Se **declara** dentro de una función y solo tiene **visibilidad** dentro de esta
- Si se declara una global y local con el mismo nombre, **la local tiene prioridad**
- **No hay problema** con variables del mismo nombre en distintas funciones
- Es conveniente usar variables con nombres distintos para evitar confusiones

Parámetros

Son los datos que van entre los () de las funciones

PASAJE POR VALOR

Se llama así cuando a función se le pasa como parámetro el valor actual de la variable

PASAJE POR REFERENCIA

Se llama así cuando a la función se le pasa como parámetro una dirección de memoria

Para validar que se halla escrito un numero

```
if(scanf("%d",&num)==1)
```

Si en este caso ingresamos un texto no devuelve nada,

Cada que recibo un puntero hay que verificar que sea distinto a NULL

Tomar todo dato como texto para poder analizarlo

- ¿Por qué utilizar funciones?
- Si no se usan distintas funciones y se escribe todo en el main:
    - El código se hace incontrolable.
    - Ante la depuración, tenemos que buscar en un código inalcanzable.
    - Usa funciones innecesarias. Programa de forma redundante.

- ¿Cómo se hace?
    1. Declaración del prototipo: Declara la función. Verifica la consistencia.

       ¿Qué va a devolver? (tipo) - ¿Cómo se va a llamar? (nombre) - ¿Qué va a recibir? (parámetro)

       `int SumarNumeros(int unNumero; int otroNumero);`
    2. Desarrollo o implementación:

       `int SumarNumeros(int unNumero; int otroNumero) {`
       `//Scope`
       `int resultado;`
       `resultado = unNumero + otroNumero;`
       `return resultado` (Permite que una función retorne un valor cuando se la llama.)
       `}`
    3. Llamada de la función declarada desde cualquier otra.
- ¿Cómo funciona?

  Cuando llamamos a la nueva función desde el *main*, siguiendo el ejemplo anterior, lo escribimos como `SumarNumero(num1, num2)` y la función main reemplaza los datos de la función a la que llama, por los datos dados en la misma función.

Ejemplo:

```c
PedirEntero(char mensaje [], int min, int max); //PROTOTIPO

int main(void){                                  //HACEMOS EL LLAMADO DE LA FUNCIÓN
  setbuf(stdout, NULL);
  int nota;
  int legajo;
  nota = PedirEntero("Ingrese su nota: ", 1, 10);
  legajo = PedirEntero("Ingrese su Legajo: ", 1, 1000);
  printf("La nota es: %d. \n", nota);
  printf("EL legajo es: %d.", legajo);
  return 0;
 }


PedirEntero(char mensaje[], int min, int max){ //HACEMOS EL DESARROLLO DE LA FUNCIÓN
  int numeroIngresado;
  printf("%s", mensaje);
  scanf("%d", &numeroIngresado);
  while(numeroIngresado < min || numeroIngresado > max){
      printf("Ingrese un número entero entre %d y %d: ", min, max);
      scanf("%d", &numeroIngresado);
  }
  return numeroIngresado;

 }
```

En este ejemplo, la función `PedirEntero` recibe una cadena de caracteres y dos enteros. Luego de declarar esta función, cuando se la llama en el main, se escribe `PedirEntero` y entre paréntesis los datos que se necesitan para que funcione (un string, un mínimo y un máximo).

Dentro de la función con el parámetro de string, hay que poner un `printf` para que muestre el mensaje especificado cuando se llama la función.

En el caso de "nota", la variable mensaje se transforma en "Ingrese su nota:", min en 1, y max en 10. Y en el caso de "legajo", mensaje es "Ingrese su legajo:", min es 1, y max es 1000.

Tipos de funciones

	Devuelve	Recibe
1	`int`	`int, int`
	Recibe valores cuando es llamada, hace algo y retorna algo. Recomendada. `variable = Sumar(1, 5)`	
2	`int`	`void`
	Dentro de la función están los valores (ya los tiene o los pide), hace algo y retorna un valor. `variable = Sumar()` → Los números a sumar ya se declararon en la función "Sumar".	
3	`void`	`int, int`
	Recibe valores, hace algo y **no retorna nada**. Se puede usar para mostrar un resultado. `Sumar(2, 8)` → La función no debe ser igualada a una variable ya que no retorna nada.	
4	`void`	`void`
	No recibe ni devuelve ningún tipo de dato. **No utiliza return**. `Sumar()` → La función no debe ser igualada a una variable ya que no retorna nada.	

Bibliotecas

Header File

- Es un archivo *.h*
- Lo utilizamos para escribir los prototipos de las funciones.
- Debemos incluir el stdio.h, para que funcione correctamente. `#include <stdio.h>`
- **Se incluyen** en el archivo *main* y el archivo *.c*. `#include "Biblioteca.h"` funciona como un copy paste

Source File

- Es un archivo *.c*
- Debe tener el mismo nombre que el Header.
- Codeamos toda la función.
- Debemos incluir el Header. `#include "Biblioteca.h"`
- **No se incluyen** en el archivo *main*. `#include "Biblioteca.c"`

Documentación de funciones

Se utiliza para dejar constancia de cómo trabaja la función.

Atajo de bloque de comentarios: `/// Enter↲`

- `@brief` descripción corta del cometido de la función
- `@param` para indicar el cometido de un parámetro
- `@returns` para indicar qué devuelve una función

Recursividad

Una función recursiva es la que se invoca a sí misma. Para obtener la solución, la función tiene que repetirse.

Razones para NO usarla:

- Crean múltiples variables en la memoria.
- Ocupan mucha memoria y tiempo de procesamiento.
- Pueden generar un *Stack OverFlow* si son mal programadas.

Menú de selección

Necesitamos:

- Una estructura repetitiva:

```c
int opcion;
do{
    printf("1. Alta Usuario.\n");
    printf("2. Baja Usuario.\n");
    printf("3. Modificación Usuario.\n");
    printf("4. Salir.\n");

    printf("Elija una opción: ");
    scanf("%d", opcion);
}while(opcion != 4);
```

La pregunta se va a repetir hasta que el usuario elija la opción 4 la cual sería “Salir”.

- Un `switch` (dentro del `do while`):

```c
switch(opcion){
case 1:
    printf("Doy de alta.\n");
    break;
case 2:
    printf("Doy de baja.\n");
    break;
case 3:
    printf("Modifico.\n");
    break;
case 4:
    printf("Salgo.\n");
    break;
default:
    printf("Elija una opción del 1 al 4.\n");
}
```

Siguiendo el ejemplo de arriba, el `switch` se encarga de ver qué opción elige el usuario.

Vectores / Arrays

Una variable que ocupa varios espacios de memoria.

Lo que importa es la dirección de memoria donde empieza la variable.

Lo representamos con un índice (ponele).

- Array[0], Array[3], Array[2]…

Lo escribimos como cualquier variable, *int números [#]*

- El número que va entre corchetes, es la cantidad de índices que va a utilizar.
  - `int numeros [5]`, da lugar para 5 valores.

Para tomar un elemento en particular del vector, se muestra de la sig. forma:

Para `int numeros[5] = {3, 6, 9, 5, 2}`:

- `numeros[2] = 9`
- `numeros[5] = Ø`
- `numeros[4] = 2`

Si en vez de escribirlo de esa forma, agregamos un "`&`" delante de la variable, muestra la dirección en memoria.

Suponiendo que la dirección de memoria del vector comienza en 100:

- `&numeros[2] = 108`
  - Ya que comienza en 100, el primer dato del vector es [0] y cada entero pesa 4 bytes, el tercer dato de la variable (`[2]`) se encuentra en 108.

Siguiendo el ejemplo anterior, hay otra forma de mostrar un vector:

- `printf("%d", números);`
  - Esto va a devolver **la dirección de memoria** del primer elemento del vector, ya que no se puede representar todos los elementos como una sola variable.

Carga secuencial

Los datos del vector se introducen con un `for`.

Si en el bucle pedimos más datos de los que puede tener el vector que declaramos (por ejemplo, damos 6 datos en vez de 5), reemplaza datos en la siguiente dirección de variable.

Carga aleatoria

Los datos del vector se introducen con do while.

El usuario puede elegir dónde cargar el dato.

Si inicializamos el vector con llaves, va a poner 0 en los índices que no haya valor.

- `int números[5] = {2, 3}` → 2, 3, 0, 0, 0.
- `int números[5] = { }` → 0, 0, 0, 0, 0.
- Si son muchos para inicializar y son iguales:
  - ```
    int numeros[100];
    for (i = 0; i < 100; i++){
    numeros[i] = 2;
    }
    ```

Recibe un vector como parámetro, y una cantidad de veces que se va a escribir.

Referencia

Pasaje por referencia: (copia dirección de memoria) → **Vectores**.

- Modifica la dirección de memoria a la que llama.

Pasaje por valor: (copia de valor)

Burbujeo

Compara un dato con el siguiente (i ~ i+1)

```
for(i = 0; i < tam-1; i++){
    for(j = i+1; j < tam; j++){
        if(listaValores[i]<listaValores[j]){
            aux = listaValores[i];
            listaValores[i] = listaValores[j];
            listaValores[j] = aux;
        }
    }
}
```

Es ineficiente.

En cambio, podemos utilizar:
```

```
if(listaValores != NULL && tam >= 0){
    do{
        flagSwap = 0;
        for(i = 0; i < tam-1; i++){
            if(listaValores[i]<listaValores[i+1]){
                flagSwap = 1;
                aux = listaValores[i];
                listaValores[i] = listaValores[i+1];
                listaValores[i+1] = aux;
            }
        }
    }while(flagSwap);
}
```

Punteros

```
int CalcularMaximo(int vector[], int tam, int* refMaximo){
    int max;
    int i;
    int flagPrimerPos = 0;
    for(i=0; i<tam; i++){
        if(vector[i]>0 && (flagPrimerPos==0 || vector[i]>max)){
            max = vector[i];
            flagPrimerPos = 1;
        }
    }
    *refMaximo = max;
    //Al valor al que apunta "maximo", lo asigna a "max"
    return flagPrimerPos;
}//Devuelve 1 o 0, si devuelve 0, no hubo positivos
```

Int* refMaximo -> Cuando se llama desde otra función, el dato que se le pasa tiene que ser una dirección de memoria de una variable local del main (u otra función); Ej.: &máximo.

Luego de que se ejecuta la función "CalcularMaximo", la variable local "máximo" del main toma el dato guardado en refMaximo de la otra función.

```
int lista[5]={-5,-9,-3,-4,-5};
int maximo;
int respuesta;
respuesta = CalcularMaximo(lista, 5, &maximo);
// var    =     funcion (vector, tam, guarda en la dir de memoria)
if(respuesta==1){// Si la función retorna 1, se ingresó.
    printf("El mayor de los positivos es %d.", maximo);
}else{
    printf("No se ingresaron positivos.");
}
```

Cadena de caracteres

En C no existe el dato tipo *string* de JavaScript. En cambio, utilizamos una **cadena de caracteres**.

Cada carácter se guarda como elemento de un array, y termina con un **\0**. Por ejemplo, para la palabra "casa", C necesita 5 elementos en el array ('c' 'a' 's' 'a' '\0').

Entonces, si queremos pedir un nombre, y queremos que sea de máximo 10 caracteres, tenemos que declarar char nombre[**11**]. La cadena debe ser de 11 caracteres, ya que el \0 utiliza uno de ellos.

El scanf ya no es de confiar, porque lee hasta que encuentra un espacio. Aquí es donde entra la función **gets**.

Funciones para cadena de caracteres

Función ***gets***

Lee directamente del teclado hasta que se presiona el Enter↲.

Argumento:

```
gets(texto);
```

NO FUNCIONA en Linux.

En Linux se utiliza la función **fgets**.

Función ***fgets***

También lee directamente del teclado, pero **agrega un \n**. Para solucionar este problema (agrega un enter al final de la cadena) lo que se puede hacer es:

Suponiendo que nuestra cadena es "Pelotero", el largo(`length`) va a ser de 10, 8 de la palabra, 1 de \n y 1 de \0:

```
cadena[strlen(cadena)-1] = '\0'
```

- Lo que hace es reemplazar el \n que se encuentra en la posición 9 (length-1) por un \0.

Argumento:

```
fgets(texto, 50, stdin);
```

- Cadena de caracteres,
- Cantidad máxima de caracteres,
- Buffer de entrada

Sirve para buscar dentro de un archivo (próximamente…)

Las siguientes funciones están **todas** definidas en la biblioteca `<string.h>`

Función strlen

No es lo mismo largo de cadena que capacidad. Una cadena puede tener capacidad de 50 caracteres y tener un largo de 3 ('A' 'n' 'a')

Para medir el largo de una cadena, hasta el \0, utilizamos la función `strlen`.

Argumento de srtlen:

`int = strlen(vector)` / `int = strlen("hola a todos")` => 12 caracteres

- Lo igualamos a un entero para luego mostrarlo por un printf.

Función strcpy

La función `strcpy` sirve para copiar una cadena de caracteres desde un origen a un destino.

Argumento:

```
strcpy(destino, origen);
strcpy(char*, const char*);
strcpy(cadena, "Perro"); // strcpy(cadena, otraCadena);
```

Función strncpy

Es esencialmente lo mismo que `strcpy`, tan solo que al argumento se le agrega la cantidad máxima de caracteres disponibles.

Argumento:

```
strncpy(destino, origen, tamaño);
strncpy(char*, const char*, int num);
```

```
strncpy(cadena, “Perro”, 3); // strcpy(cadena, otraCadena, int);
```

- `cadena = “Per”`

Función strcmp

Permite comparar dos cadenas.

Argumento:

```
strcmp(cad1, cad2);
strcmp(char*, const char*);
strcmp(cadena, “Perro”); // strcmp(cadena, otraCadena);
```

Si la función devuelve:

- **menor a cero**, *cad1* es menor que *cad2* (*cad1* viene después que *cad2*).
- **cero**, *cad1* es la misma que *cad2*.
- **mayor a cero**, *cad1* es mayor que *cad2* (*cad1* viene antes que *cad2*).

Función stricmp / strcmpi

Son esencialmente lo mismo que `strcmp`, tan solo que ignoran las mayúsculas y las minúsculas.

Función strupr / strlwr

Modifican el case de las cadenas para hacerlas completamente mayúsculas y minúsculas respectivamente.

Argumento:

```
strupr(cadena);
strlwr(cadena);
```

Trabajan por referencia.

*Función **Fopen***

La función Fopen() utiliza dos parámetros. FILE *fopen(char*, char*); El primer parámetro es el nombre externo del fichero a abrir. Escrito en el formato soportado por el sistema operativo.

El segundo parámetro es una cadena de caracteres que describe el uso al que se va destinar el fichero (Modo de apertura)

Funcion fclose

Esta función cierra un archivo abierto por la función fopen(). Además, realiza el vaciado del buffer que podría haber quedado parcialmente lleno al momento de cerrar el fichero.

Función Sizeof

La función recibe como único parámetro o el nombre de una variable, o el nombre de un tipo de datos, y devuelve su tamaño en bytes. De esta forma, sizeof(int) devuelve el número de bytes que se utilizan para almacenar un entero. La función se puede utilizar también con tipos de datos estructurados

El operador sizeof devuelve el tamaño de una variable o tipo de dato durante la compilación, no durante la ejecución del programa. Veamos algunos ejemplos:

sizeof(int) devuelve el valor 2 en los sistemas operativos de 16 bits y 4 en los de 32 bits.

Si tenemos char a[20], sizeof(a) devuelve el valor 20, y si tenemos float a[6], sizeof(a) devuelve el valor 24 (4*6).

Funcion Fwrite

La función fwrite() escribe cuenta número de elementos, cada uno de ellos de tam bytes de longitud, del array apuntado por buf al archivo asociado a la variable f. El indicador de posición del archivo se incrementa en el número de bytes escritos. La función fwrite() devuelve el número

de elementos realmente escritos. Si se escriben menos elementos de los pedidos en la llamada se produce un error. La función fwrite() funciona de forma correcta en archivos abiertos en modo binario; en archivos abiertos en modo texto, pueden producirse ciertos cambios de caracteres (salto de carro seguido de salto de línea se convierte en salto de línea, etc.).

fwrite #include int fwrite(const void *buf,size_t tam,size_t cuenta,FILE *f);

*Función **strcat***

Se utiliza para concatenar cadenas. Utilizamos esta función ya que en C no se puede utilizar el +.

Argumento:

```
strcat(destino, origen);
strcat(char*, const char*);
```

- `strcat(cadena, "Perro");` // `strcat(cadena, otraCadena);`

Funciones atoi / atof

Convierte una cadena a su valor numérico entero y a flotante respectivamente.

Argumentos:

```
valor = atoi("123");
valor = atof("123.85");
```

Funciones para un solo carácter

Función isspace

Revisa si el carácter es un espacio en blanco o no, caracteres tales como:

- `' '` espacio
- `'\n'` salto de línea
- `'\t'` tabulación horizontal
- `'\v'` tabulación vertical
- `'\f'` salto de página
- `'\r'` retorno de carro

Si el carácter es alguno de estos, la función retorna un número distinto de 0, si no lo es, retorna 0.

Está definido en la biblioteca `<ctype.h>`

Función isdigit

Revisa si el carácter es de tipo numérico.

Si el carácter es un entero, la función retorna 1, si no lo es, retorna 0.

Está definido en la biblioteca `<ctype.h>`.

Función isalpha

Revisa si el carácter está en el alfabeto (a-z & A-Z).

Si el carácter es parte del alfabeto, la función retorna un número distinto de 0, si no lo es, retorna 0.

No toma la 'Ñ'.

Está definido en la biblioteca `<ctype.h>`.

Sentencia Return

Antes de empezar la explicación de las funciones, conviene explicar la sentencia return. La sentencia return permite, en primer lugar, salir de la función desde cualquier punto de la misma, y

en segundo lugar, devolver un valor del tipo de la función, si ello es necesario (no se devuelve ningún valor si la función es de tipo void).

```
int Comparacion(int a,int b)
{ if (a>b) return 1;    // a es mayor que b
  if (a<b) return -1;     // a es menor que b
  return 0;               // a y b son iguales */
}
```

Como se observa en el ejemplo, una función puede contener más de una sentencia return. Ello permite, la posibilidad de poder salir de la función desde distintos puntos de la misma. Un aspecto que conviene resaltar es el hecho de que una función también termina su ejecución si llega al final de la misma sin encontrar ninguna sentencia return.

Ello es posible en toda función de tipo void.

```
void A(int *a) { *a=5; }
```

Vectores paralelos

Vectores en donde cada componente de un vector, va a ser correlativa a cada componente del vector (x[0] = y[0] = z[0]).

Matriz = vector bidimensional.

Se definen como un array, pero con dos [ ].

```
char nombre[tam de fila][cant de columnas]
```

Filas: cant de nombres

Columnas: cant de caracteres para el nombre

Estructuras

`Typedef`: instrucción que nos permite definir un tipo de dato

```
typedef struct{
//Variables que queremos que sean parte de la estructura
int nota;
int edad;
char nombre[50];
}eAlumno;
```

Esto funciona para luego utilizarlo como tipo de variable.

Esta estructura va a representar a un alumno solo, con esos 3 datos.

Declaramos la variable:

```
eAlumno unAlumno = {9, 25, "Pepe"};
```

Imprimimos el mensaje:

```
printf("Nombre: %s, Edad: %d, Nota: %d.", unAlumno.nombre, unAlumno.edad, unAlumno.nota);
```

Las estructuras trabajan por valor.

Anidamiento de estructuras / Composición

Utilizar datos de tipo de una estructura dentro de otra.

Estructuras de fecha

```
typedef struct{
    int día;
    int mes;
    int año;
}eFecha;
```

Normalización de datos

El proceso donde pienso qué datos voy a manejar, y pregunto ¿Cómo puedo hacer para que haya la menor redundancia posible? **Empleando Id's**.

> En vez de tener dos datos que van a repetirse dentro de una estructura, puedo crear otra estructura con estos datos y llamarlos con IDs:

Alumno

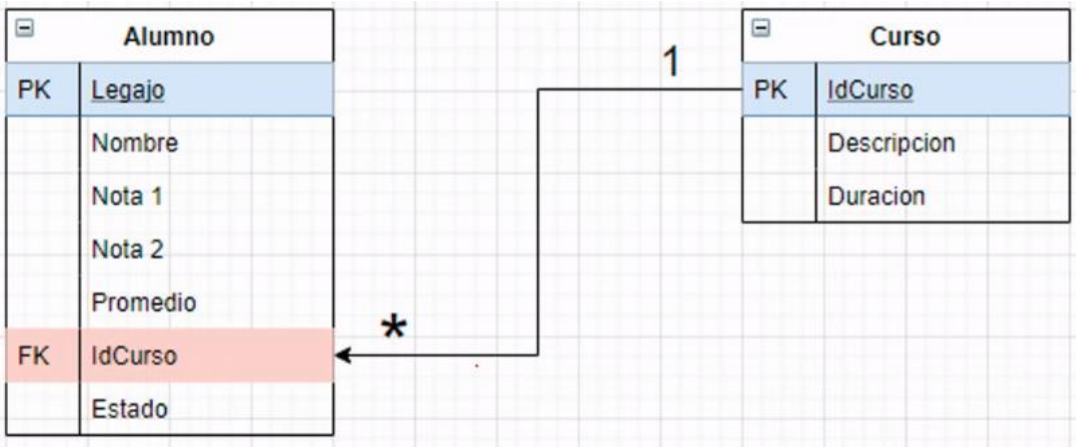
Legajo	Nombre	Nota 1	Nota 2	Promedio	IdCurso
1	Maria	3	5	4	100
2	Ana	4	5	4,5	101
3	Juan	9	6	7,5	102
4	Pedro	8	7	7,5	101
5	Anibal	5	8	6,5	101
6	Mariana	4	9	6,5	100
7	Alan	8	3	5,5	100
8	Sol	9	5	7	101
9	Daniela	4	2	3	102
10	Mariano	2	4	3	100
11	Octavio	3	4	3,5	101
12	Marina	5	6	5,5	100
13	Christian	6	7	6,5	102
14	Mauricio	7	8	7,5	101
15	Camila	4	9	6,5	100
16	Maximilian	8	6	7	102

Curso

IdCurso	Curso	Duracion
100	Java	10
101	Inkscape	15
102	Office	7

Diagrama entidad relación

Siguiendo el ejemplo anterior, relacionamos la tabla Alumno con la tabla Curso.



PK (Primary Key): Identifica un dato dentro de un conjunto.

FK (Foreign Key): Clave prestada, es una PK en otra entidad.

Relacionar ambas estructuras

Esto, luego se puede utilizar para mostrar en forma de lista (sin `switch`).

Necesitamos 1 `for` para recorrer la primera lista, y un segundo `for` para recorrer la segunda (cada con su respectivo tamaño).

EJEMPLO

Dentro del segundo `for` hacemos la comparación. Tomamos el ID del curso ingresado en la estructura de alumnos, y lo comparamos con los ID en la estructura de cursos. Cuando lo encuentre, lo mostramos o hacemos lo que se necesite, y un `break` para que deje de buscar cuando encuentre.

`AAAAAAAAAAAAAAAAA`

Punteros

Comienzo de punteros.

Todo dato se almacena en la RAM, dividida en distintas celdas numeradas, que conocemos como dirección de memoria. Estas direcciones no las conocemos, ya que son muchísimas y se encarga de buscarlas el SO; pero principalmente, porque son random (RAM: Random Access Memory).

Segmento de pila

Cada vez que se llama a una función entra en este segmento con toda su información y allí se guardan:

- Los llamados a las funciones
- Los parámetros de las funciones
- Las variables locales
- Otra información necesaria para el funcionamiento del programa.

Stack

Al intercambiar variables, como ya vimos, podemos utilizar el `swap` (valorA, valorB, aux). Al hacer el intercambio llamando a una función, intercambia **los valores** dentro de la función que llama, una copia de esos valores, que **en el main no tiene ningún impacto**.

Esto se soluciona con *punteros*. Si esos valores los pasamos como punteros, la función modificaría **las direcciones de memoria** del dato.

`*` = Operador de inacceso

Al declarar un puntero que por ahora no va a tener ningún valor, no podemos dejarlo sin declarar, vamos a utilizar el `NULL`.

`NULL` = Dirección de memoria que no existe.

Para asignar un puntero a una variable, se hace de la siguiente forma:

```
int* p = NULL;
int number = 88;
 &numero;
```

Esto lo que hace, es copiar la dirección de memoria de 'numero', no su valor.

Entonces, a la hora de utilizar ese nuevo dato, si hacemos

```
printf("%d", numero); //Valor de 'numero'.
printf("%d", &numero); //Dirección de memoria de 'numero'.
printf("%d", p); //Dirección de memoria de lo que apunta el puntero.
printf("%d", *p); //Valor al que apunta el puntero.
        Si el puntero está como 'NULL', y pedimos que muestre el valor
        al que apunta, rompe el programa.
printf("%d", &p); //Dirección de memoria del puntero.
```

Todo puntero vale 4 bytes, ya que lo que guarda es una dirección de memoria (`int`).

Aritmética de punteros

Podemos utilizar operadores aritméticos (+, -, ++, --). Estos operadores afectan directamente a la dirección de memoria a la que apunta el puntero, pero afectan a cada dato de manera diferente:

	Posición Actual	a++	a--	a=a+2	a=a-3
char *a	0xA080	0xA081	0xA07F	0xA082	0xA07D
int *a	0xB080	0xB084	0xB07C	0xB088	0xB06E
float *a	0xC080	0xC084	0xA07C	0xA088	0xA06E

Al asignar un puntero a un array, hay 3 formas de hacerlo:

```
array[5] = {4, 25, 32, 14, 10};
pArray;

     pArray = array;
     pArray = &array;
     pArray = &array[0];
```

Las tres apuntan a lo mismo, la **dirección de memoria del primer dato** del array.

Pero si escribimos `pArray = array[0];` estaríamos asignando el **valor** del primer dato del array.

A la hora de mostrar los datos de un array a través de un puntero, debemos utilizar el operador '+' y el operador de inacceso (*), de la sig. forma:

```
for(int i = 0; i < 5; i++){
   printf("%d\n", *(pArray+i));
```

De esta forma, busca el valor al que apunta '`pArray`' sumándole `i`, que al ser un entero, suma de a 4. Entonces busca el primer valor, luego el segundo y así…

Por el otro lado, si escribimos `printf("%d", *pArray+i);` lo que va a hacer, es buscar el primer valor al que apunta y sumarle i.

Punteros a Estructuras

Al momento de utilizar un puntero de tipo estructura, la forma de imprimir los campos es de la siguiente forma:

```
for(int i = 0; i < 5; i++){
     printf("%d\n", (*(pArray+i)).campo);
```

Así, estaríamos llamando al valor de tipo de dato estructura '`pArray`', y de este tomamos el dato del campo que llamamos.

Hacerlo así no es recomendado. Así que una mejor forma es utilizar el operador flecha (—>).

Se utiliza para acceder a través del puntero hacia un atributo, a la estructura que apunta el puntero.

```
for(int i = 0; i < 5; i++){
   printf("%d\n", (pArray+i)->campo);
```

Punteros a Función

Un puntero a función es una variable que almacena la dirección de una función. Este tipo de construcción es útil pues encapsula comportamiento, que puede ser llamado a través de un puntero. Veamos cómo funciona mediante un ejemplo sencillo que crea un puntero a una función de imprimir y lo invoca.

```
#include <stdio.h>
#include <stdlib.h>
void saludar()
{
printf("Hola\n");
}
int main(void)
{
void (*pFuncion)(void);
pFuncion = saludar;
pFuncion();
return EXIT_SUCCESS;
}
```

Puntero a funciones con parámetros

Si la función a la que se quiere apuntar recibe algún parámetro, hay que indicar sus tipos al declarar el puntero. En el siguiente ejemplo la función recibe un parámetro por valor, otro por referencia y muestra sus valores. Cuando se declara el puntero se escriben los tipos de los argumentos entre paréntesis.

```
#include <stdio.h>
#include <stdlib.h>
void mostrar(int parametroA, float* pParametroB)
{
  printf("A: %d - B: %f \n",parametroA,*pParametroB);
}

int main(void)
{
  float auxiliarFloat = 3.14;
  void (*pFuncion)(int,float*);
  pFuncion = mostrar;
  pFuncion(22,&auxiliarFloat);
  return EXIT_SUCCESS;
}
```

Puntero a función como parámetro de otra función

Las funciones también pueden pasarse como parámetros a otras funciones. En

el siguiente programa se define una función principal “Calcular” que recibe tres parámetros, los dos operandos y el puntero a la función que realiza la operación.

Con este ejemplo puede verse que los punteros a funciones sirven para eliminar la redundancia en el código: al pasar la función como parámetro nos ahorramos tener que escribir variantes de la función principal para realizar cada una de las operaciones.

```
 #include <stdio.h>
 #include <stdlib.h>
void sumar(int parametroA, int parametroB, int* pResultado)
{
    *pResultado = parametroA + parametroB;
}

void restar(int parametroA, int parametroB, int* pResultado)
{
   *pResultado = parametroA - parametroB;
}

int calcular( int parametroA, int parametroB, void(*pFuncion)(int,int,int*) )
{
   int auxResultado;
   pFuncion(parametroA , parametroB , &auxResultado);
   return auxResultado;
}

int main(void)
{
  int auxiliar;
  auxiliar = calcular(10 , 5 , restar);
  printf("El resultado de la resta es %d\n",auxiliar);
  auxiliar = calcular(10 , 5 , sumar);
  printf("El resultado de la suma es %d\n",auxiliar);

  return EXIT_SUCCESS;
}
```

Memoria dinámica

Toda variable ocupa espacio, el cual no puede ser modificado en tiempo de ejecución. La memoria tiene distintos segmentos:

- **Segmento de código**
  - Todas las instrucciones de una función.
- **Memoria estática**
  - Variables globales.
- **Pila / Stack**
  - Llamados de funciones
  - Parámetros
  - Variables locales
  - Es un segmento muy pequeño
- **Montón / Heap** (memoria dinámica)
  - Todo lo que se guarda va a tener un comportamiento dinámico.
  - No sabemos cuánto tiempo va a estar el dato en memoria.
  - Tiene direcciones de memoria
- **Espacio libre**
  - Entre el Heap y la Pila. A través de este se van moviendo.

Cuando devolvemos un puntero a través de una función, y luego la llamamos desde otra, la dirección de memoria a la que apunta se guarda en la pila, la cual pisa los datos al momento de salir de dicha función, por lo tanto, no funciona tan solo hacer un llamado y ya; debemos hacer que el valor al que apunta se guarde en el heap, cuyos datos existen indetermidamente. Aquí es donde entra:

Constructores

Funciones que inicializan datos, a través de memoria dinámica

Por defecto

- No reciba nada y devuelve un puntero con un valor por defecto.

Parametrizados

- Recibe como parámetros los valores con los que se van a inicializar las estructuras

Función `malloc`

Reserva espacio en el Heap y devuelve la dirección de memoria que encuentra libre.

```
(tipo_de_dato*) malloc(espacio a reservar)
```

Normalmente usado con `sizeof()`

Se encuentra en `<stdlib.h>`

Si no encuentra espacio en memoria, malloc devuelve `NULL`, entonces, SIEMPRE tenemos que validar que no lo sea.

Para reservar espacio en memoria para un vector, debemos multiplicar el tamaño del tipo de dato por la cantidad máxima de elementos que queremos:

```
tipo_de_dato* array;
array = (tipo_de_dato*) malloc(sizeof(tipo_de_dato)*TAM);
```

Esto es equivalente a escribir `int array[5];` tan solo que utilizado malloc, guarda el array en el Heap y puede ser expandido, de la otra forma, no.

Función `realloc`

Se utiliza para modificar el tamaño de un array.

```
realloc(puntero_a_reacomodar, tamaño_de_cant_total_de_datos);
```

Al cambiar el tamaño del array, lo que hace la función es buscar esa cantidad indicada, en forma de espacios vacíos consecutivos del Heap. Si la función no encuentra espacio, retorna `NULL`.

Para evitar que realloc borre los datos ya guardados por el retorno de `NULL`, guardamos lo que retorna en un auxiliar, y luego lo pasamos:

```
tipo_de_dato* array;
array = (tipo_de_dato*) malloc(sizeof(tipo_de_dato) *TAM);
aux = (tipo_de_dato*) realloc (array,
sizeof(tipo_de_dato)*NUEVO_TAM);
if(aux != NULL){
      array = aux;
}
```

Función ***free***

Libera el espacio de memoria que ocupa el puntero.

Tenemos que fijarnos que ese espacio no lo utilicemos en otra parte, porque si el auxiliar que deseamos liberar, apunta a los mismos datos que una variable que vamos a utilizar, se borran los datos de ambas.

LinkedList

El LinkedList es una estructura que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena es dinámico, es decir, que no es necesario declarar su tamaño como pasa con los Arrays. Los LinkedList nos permiten añadir, eliminar y modificar elementos de forma transparente para el programador.

¿Qué contiene la LinkedList?

Posee un '`.a`', un '`.c`' compilado, podemos ver la herramienta, pero no ver cómo funciona.

No necesita elementos de memoria consecutivos, trabaja con nodos. El nodo representa el vagon de un tren, apunta a datos y al nodo2, el nodo2 conecta con su dato y con el 3, y así sucesivamente.

- Crear una lista
    - Crear el puntero
- Llamar a la función ll_newLinkedList()
    - Creamos un empleado

```
#ifndef__LINKEDLIST
#define__LINKEDLIST
struct Node
{
void* pElement;
struct Node* pNextNode;
}typedef Node;

struct LinkedList
{
Node* pFirstNode;
Int size;
} typedef LinkedList
#endif
```

Archivos

Puntero de tipo `FILE*`. Conecta con el archivo.

Para manipular el archivo hay 3 pasos:

1. Abrir el archivo
2. Usarlo
3. Cerrarlo

Hay dos formas para trabajar con un archivo:

1. Modificar los datos directamente del archivo.
2. Traer los datos por memoria.

Split de la cadena: cortar la cadena en pedazos (parsear)

Formas de abrir un archivo

Modo	Detalle
`r`	Abre un archivo de texto para operaciones de lectura.
`w`	Abre un archivo de texto para operaciones de escritura
`a`	Abre un archivo de texto para añadir datos.
`rb`	Abre un archivo binario para operaciones de lectura.
`wb`	Abre un archivo binario para operaciones de escritura.
`ab`	Abre un archivo binario para añadir datos.
`r+b`	Abre un archivo binario para operaciones de lectura escritura.
`w+b`	Abre un archivo binario para operaciones de lectura escritura.
`a+b`	Abre un archivo binario para operaciones de lectura escritura.
`r+`	Abre un archivo de texto para operaciones de lectura escritura.
`w+`	Abre un archivo de texto para operaciones de lectura escritura

Cada función de la biblioteca cuenta con un Test unitario asociado mediante el cual se podrá verificar el correcto funcionamiento de la misma.

```
startTesting(1); // ll_newLinkedList
startTesting(2); // ll_len
startTesting(3); // getNode - test_getNode
startTesting(4); // addNode - test_addNode
startTesting(5); // ll_add
startTesting(6); // ll_get
startTesting(7); // ll_set
startTesting(8); // ll_remove
startTesting(9); // ll_clear
startTesting(10); // ll_deleteLinkedList
startTesting(11); // ll_indexOf
startTesting(12); // ll_isEmpty
startTesting(13); // ll_push
startTesting(14); // ll_pop
startTesting(15); // ll_contains
startTesting(16); // ll_containsAll
startTesting(17); // ll_subList
startTesting(18); // ll_clone
startTesting(19); // ll_sort
```