

# Programación Imperativa

**CUANDO ESCRIBÍ ESTE CÓDIGO,  
SÓLO DIOS Y YO SABÍAMOS  
CÓMO Y PARA QUÉ LO HICE**



**AHORA, SÓLO DIOS LO SABE**



**Atribución - No Comercial (by-nc):** Se permite la generación de obras derivadas siempre que no se haga con fines comerciales. Tampoco se puede utilizar la obra original con fines comerciales. Esta licencia no es una licencia libre.

# UNIDAD 2

# TEMAS

**Estructuras de datos:** en esta unidad nos ocuparemos de una estructura de datos concreta: los **arreglos estáticos**.

Detalles de implementación y principales algoritmos para manipular arreglos:

✦ *insertar elementos al final*, ✦ *insertar elementos ordenados*, ✦ *recorrer la estructura*, ✦ *buscar elementos*, ✦ *eliminar elementos*.

También veremos cómo podemos **definir nuestros propios tipos de datos** para luego almacenar datos de estos tipos en arreglos.

# Estructuras de datos

# Estructuras de datos

Una **estructura de datos** es una forma de almacenar varios datos.

Cuando se requiere guardar **varios** datos que tienen sentido juntos, es necesario contar con una estructura que los almacene, un contenedor.

Existen muchas estructuras de datos que suelen estudiarse: arreglos, listas enlazadas, árboles, grafos, tablas de hashing, etc. En esta asignatura sólo usaremos arreglos y listas enlazadas simples.

Estas estructuras de datos suelen ser la base de bajo nivel para construir otros contenedores, que se utilizan luego a alto nivel (por ejemplo, los vectores en C++), ya que permiten manejar la memoria con alta eficiencia.

# Estructuras de datos: clasificación

De acuerdo al **tipo de datos** que contiene, una estructura puede ser:



## Heterogénea

Si los datos que la componen pueden ser de distinto tipo.



## Homogénea

Si los datos que la componen deben ser todos del mismo tipo.

# Estructuras de datos: otra clasificación

De acuerdo a la **ocupación de memoria**, una estructura puede ser:



## Estática

La **cantidad de elementos** que contiene es **fija**, es decir: la cantidad de memoria que ocupa no cambia durante la ejecución del programa.

## Dinámica

La **cantidad de elementos** que contiene es **variable**, es decir: la cantidad de memoria que ocupa puede cambiar durante la ejecución del programa.

# Estructuras de datos: otra clasificación

De acuerdo al **acceso** a sus **elementos**, una estructura puede tener acceso:



## Secuencial

Si para acceder a un elemento se debe respetar un orden (por ejemplo: **pasando por todos los elementos que lo preceden**).



## Directo

Si se puede **acceder directamente** a un elemento particular (ejemplo: indicando su posición).



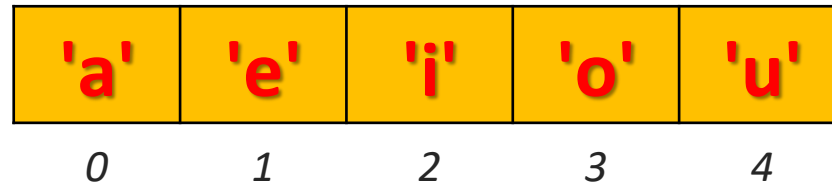
# Arreglos estáticos

## (de tamaño fijo)

# Arreglos estáticos

Un arreglo es una *estructura de datos* formada por una secuencia de elementos **homogéneos** (del mismo tipo), donde cada elemento tiene una posición relativa, determinada por un **índice**, que comienza en el 0.

Suelen denominarse con la palabra en inglés **array**.



# Arreglos estáticos: características

- ▶ Un arreglo es una colección de datos relacionados de igual tipo e identificados bajo un nombre genérico único.
- ▶ En C++ se pueden crear arreglos de cualquier tipo de datos (simples o estructurados).
- ▶ Se debe establecer, en la declaración, la cantidad de elementos (dimensión física) que puede tener como máximo el arreglo.
- ▶ La estructura completa ocupa un segmento de memoria único y sus elementos se almacenan en forma contigua.
- ▶ Para procesar un elemento del arreglo, se debe especificar su nombre y uno o más índices (según el tipo de arreglo) que identifiquen la posición del elemento.

# Arreglos estáticos: características

- ▶ El índice es un número de tipo entero. Entonces se puede emplear como índice cualquier expresión que arroje un entero dentro del rango establecido (dimensión) para dicho índice.
- ▶ El índice que determina la posición de los elementos de un arreglo comienza siempre con cero.
- ▶ C++ admite operar fuera del rango preestablecido por la dimensión (no arroja error), pero con resultados impredecibles, ya que se accede a memoria que no corresponde al arreglo y podrían modificarse datos que estuvieran almacenados en ese espacio.

# Arreglos estáticos: clasificación

Según el número de índices que determinan la posición de un elemento en el arreglo, podemos definir el tipo de arreglo:

## Arreglos lineales o unidimensionales:

La posición de un elemento la determina un único índice. También son llamados *vectores*.



# Arreglos estáticos: clasificación

## Arreglos bidimensionales:

Se requieren 2 índices para posicionar un elemento en la estructura. También son conocidos como *matrices* ó *tablas*.

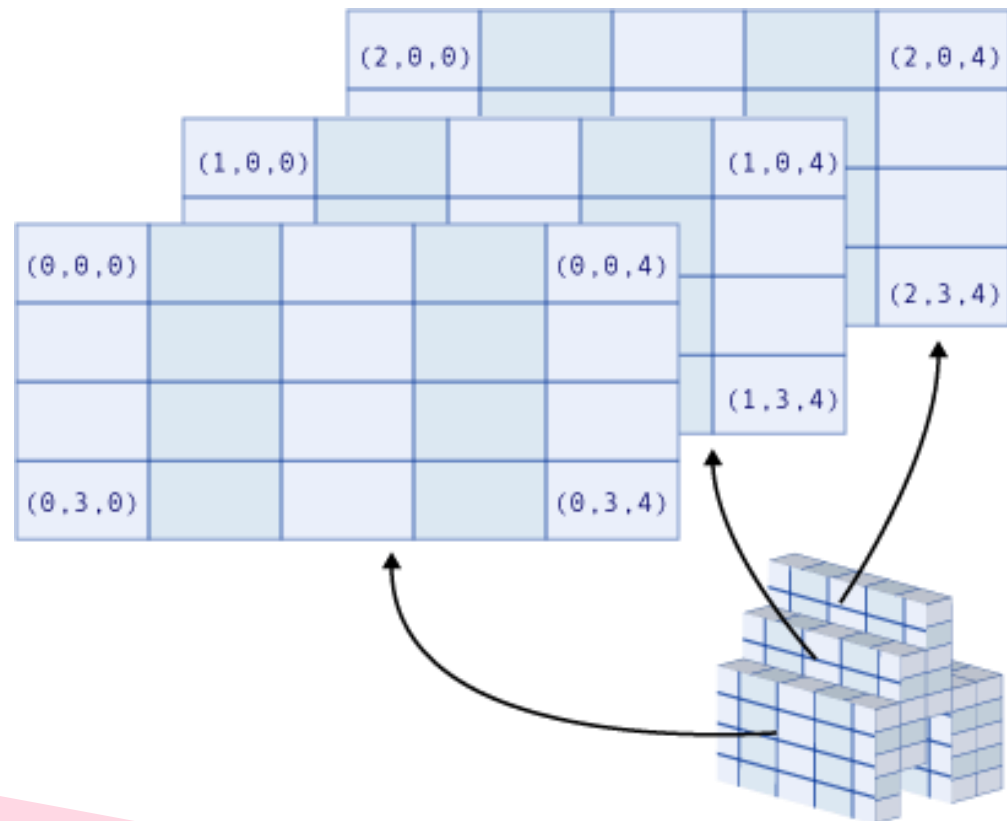
(0,0)				(0,4)
(1,0)				
(3,0)				(3,4)

Los elementos se identifican por su número de fila y de columna, siendo el primero el que está en la fila 0, columna 0.

# Arreglos estáticos: clasificación

## Arreglos multidimensionales:

Se requieren más de 2 índices para posicionar un elemento en la colección. También conocidos como matrices (ó tablas) *multidimensionales*.



# Arreglos estáticos: características

Cuando se declara un arreglo (cualquiera sea su cantidad de dimensiones), se debe indicar el **tamaño** de dicho arreglo, y éste permanecerá **constante** durante el ciclo de vida del programa o de la función, es decir, no podrá cambiar su tamaño de manera dinámica (por esto es que son arreglos de tamaño estático): si el espacio es insuficiente o sobra durante la ejecución del programa, no hay nada que hacer (más que modificar el código y volver a correr el programa), ya que el tamaño es **fijo**.



# Arreglos estáticos: índices

Se pueden usar tantas dimensiones (índices) como queramos; el límite lo impone sólo la cantidad de memoria disponible. En esta materia trabajaremos sólo con arreglos de 1 y 2 dimensiones.

Los índices son cualquier expresión **entera**, entre 0 y  $\langle \text{número de elementos} \rangle - 1$ .

Es decir: si un arreglo tiene 100 elementos, el índice comenzará en 0 y terminará en 99, en números correlativos.

Esto es muy importante y hay que tener mucho cuidado, porque **el compilador no comprueba si los índices son válidos** y nos permite modificar elementos fuera del rango.

# Arreglos estáticos: declaración

Un arreglo se declara indicando: el tipo de sus elementos, el identificador de la variable y, entre corchetes, la cantidad máxima de elementos que podrá contener cada dimensión.

```
<tipo> <identificador>[<num_elemento>][[<num_elemento>]...];
```

Ejemplo de arreglo de 1 dimensión:

```
int un_arreglo[10];
```

Esta declaración hace que el compilador le indique al sistema operativo que reserve espacio suficiente para almacenar 10 valores enteros. El sistema operativo le asigna al arreglo una porción consecutiva de memoria que es múltiplo del tamaño del tipo de datos que almacena el arreglo (si un int ocupa 4 bytes, este arreglo ocupa 40).

# Arreglos estáticos: declaración

Ejemplos de arreglos de más de una dimensión:

```
int Matriz[10][10];  
  
int DimensionN[4][15][6][8][11];
```

En el caso de matrices bidimensionales, la dimensión que se coloca en primer lugar indica la cantidad de filas y, la que se coloca en segundo lugar, indica la cantidad de columnas.

# Arreglos estáticos: inicialización

Al declarar un arreglo, podemos inicializarlo con valores:

```
int arreglo[5] = {10, 20, 30, 40, 50};
```

En el caso de matrices bidimensionales, en memoria se almacenan como un “arreglo de arreglos” y, para inicializarlas, debemos indicar los valores a almacenar en las filas y columnas:

```
int matriz[2][3] = { {1,3,5}, {5,10,2} };
```

# Arreglos estáticos: inicialización

Si, al declarar un arreglo de más de una dimensión también se lo inicializa, no es obligatorio especificar el tamaño para la primera dimensión. En estos casos, la dimensión que queda indefinida se calcula a partir del número de elementos en la lista de valores iniciales. El compilador sabe contar y puede calcular el tamaño necesario de la dimensión para contener el número de elementos especificados.

```
float R[10] = {2, 32, 4.6, 2, 1, 0.5, 3, 8, 0, 12};
```

```
float S[] = {2, 32, 4.6, 2, 1, 0.5, 3, 8, 0, 12};
```

```
int N[] = {1, 2, 3, 6};
```

4 elementos

```
int M[2][3] = {{1, 3, 5}, {5, 10, 2}};
```

2 filas y 3  
columnas

```
int J[][3] = {213, 32, 32, 32, 43, 32, 3, 43, 21};
```

El número de  
elementos es 10,  
ya que hay diez  
valores en la lista.

Divide los 9  
elementos por  
3 y deduce que  
hay 3 filas

# Arreglos estáticos: inicialización

```
int M[][3] = { {213, 32, 32}, {32, 43, 32}, {3, 43, 21} };
```

equivale a:

```
int M[3][3] = { {213, 32, 32},  
                {32, 43, 32},  
                {3, 43, 21} };
```

Aquí la dimensión faltante será 3, ya que hay 9 valores y la segunda dimension es 3, entonces:  $9/3=3$ .

# Arreglos estáticos: ocupación de memoria

Si el arreglo se declara en `main{ }`, durante toda la vida o ejecución del programa estará ocupando la porción de memoria necesaria y ese recurso no podrá ser empleado en otra cosa. Si se lo declara dentro de una función o un bloque, ocupará memoria mientras el control de ejecución opere dentro de dicha función o bloque.

```
void funcion() {  
    int arreglo[10];  
    .....  
}
```

El arreglo se elimina al finalizar la función.

```
int main() {  
    do {  
        int arreglo[10];  
        .....  
    } while (.....);  
}
```

El arreglo se elimina al finalizar el bucle do-while.

# Arreglos estáticos: ocupación de memoria

Los elementos del arreglo se almacenan en posiciones consecutivas de memoria, lo que permite tener un índice.

Cada elemento ocupará el espacio que el compilador destine al tipo de dato del arreglo. Por ejemplo:

```
int numeros[5];
```

Si se trabaja con una implementación de C++ donde las variables de tipo int ocupan 4 bytes de memoria, este arreglo ocupará 20 bytes, ya que  $5 \times 4 = 20$ .

Si, al ejecutar el programa, el sistema asigna la dirección de memoria 1000 a la variable indizada numeros[0], entonces La dirección de memoria de la variable indizada numeros[3] será 1012.



# Arreglos estáticos: ocupación de memoria

Todos los arreglos se almacenan con sus elementos en posiciones consecutivas de memoria. La posición 0 del índice siempre referencia a la dirección de memoria donde comienza el arreglo.

```
int A[5] = {10, 20, 30, 40, 50};
```

10	20	30	40	50
0	1	2	3	4

En memoria:

Elemento	Dirección relativa
10	A+0
20	A+1
30	A+2
40	A+3
50	A+4

Como cualquier otra variable, A hace referencia a una dirección de memoria

# Arreglos estáticos: ocupación de memoria

Un arreglo bidimensional puede considerarse un “arreglo de arreglos”, y se almacena en posiciones consecutivas:

```
int M[2][3] = { {1,3,5}, {5,10,2} };
```

	0	1	2
0	1	3	5
1	5	10	2

En memoria:

Elemento	Dirección relativa
1	M+0
3	M+1
5	M+2
5	M+3
10	M+4
2	M+5

# Arreglos estáticos: acceso a elemento

Para acceder a un elemento en particular se deben indicar el nombre del arreglo (identificador de la variable) y la posición del índice donde se encuentra el elemento.

Las posiciones del índice siempre son números (ya sea literales, contenidos en variables o resultado de una expresión).

Ejemplos:

```
char V[5] = { 'a', 'e', 'i', 'o', 'u' };  
int M[2][3] = { {1,3,5}, {5,10,2} };
```

```
cout >> V[2];
```



```
cout >> M[0][1];
```

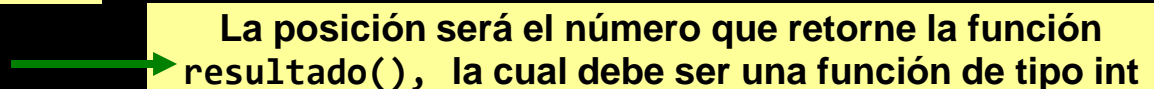


```
int i=5;
```

```
cout >> V[i-1];
```



```
cout >> V[resultado()];
```



# Arreglos estáticos: dimensión lógica

Si los elementos del arreglo contienen datos útiles o no, es una cuestión que sólo interesa al programador: para la máquina **todo** el espacio de memoria destinado al arreglo (**dimensión física**) está ocupado. Para determinar qué parte contiene datos válidos usamos el concepto de **dimensión lógica**.

La dimensión lógica no es más que **la cantidad de elementos útiles contenidos en un arreglo** (siempre comenzando desde el 0 y considerando que los elementos se almacenan en posiciones consecutivas). Usualmente la representamos mediante una variable de tipo entero, que acompaña al arreglo.

# Arreglos estáticos: dimensión lógica

La dimensión lógica nunca podrá ser mayor que la física, porque la física es el máximo de elementos que podemos guardar en un arreglo determinado.

Al declarar un arreglo debemos dar una dimensión física (a menos que lo hayamos inicializado, como ya se vio anteriormente), pero es muy importante que también declaremos **una variable** más, para indicar la **dimensión lógica**. Esta dimensión lógica **se inicializará siempre en 0**.

```
int numeros[100];  
int dimension_logica = 0;
```

Cada vez que agreguemos un elemento, la dimensión lógica se **incrementará**. De igual forma, al eliminar elementos, la dimensión lógica se **decrementará**.

# Arreglos estáticos: dimensión lógica

La máquina desconoce cuántos elementos del arreglo "le interesan" al usuario: sólo sabe cuánto espacio ocupa el arreglo. Asimismo, el usuario desconoce cuánto espacio ocupa el arreglo: sólo sabe cuántos elementos ha indicado al programa que debe guardar.

Entonces, el usuario tiene una visión de "alto nivel" donde sólo le interesa que sus datos se almacenen. En el código usamos la dimensión lógica para indicar esto, cuando sabemos que el usuario podría no utilizar todo el espacio que tiene disponible el arreglo.

En Python se utiliza algo similar a los arreglos: las listas. Pero como las listas de Python son de alto nivel, el manejo de cuántos elementos guardamos y de cuánto espacio está realmente ocupado nos es transparente. Eso no significa que no exista: sólo que no lo vemos, como sí lo hacemos con los arreglos estáticos de C++.

# Arreglos estáticos: dimensión lógica

Sólo en casos puntuales un arreglo no necesitará una dimensión lógica, y estos son cuando el arreglo siempre estará lleno en su totalidad (es decir: dimensión lógica y dimensión física coinciden). Por ejemplo: un arreglo que guarde los nombres de los 7 días de la semana.

A menos que sepamos con seguridad que ese es el caso, **siempre** deberemos declarar una **dimensión lógica**, porque nos permitirá llenar el arreglo hasta la cantidad de elementos deseados y no necesariamente hasta el final. Es decir: nos permite abstraer al usuario de qué tipo de estructura estamos usando para almacenar sus datos.



# Arreglos estáticos: dimensión física

Sabemos que la **dimensión física** de un arreglo no cambia en toda la ejecución del programa, y que nos indica el "tope" o la cantidad máxima de elementos que podemos almacenar, por lo que es posible que necesitemos usarla en algunas funciones. Por esto es útil declarar a la dimensión física como **constante**, al inicio del programa:

```
#include <iostream>
using namespace std;

const int DIMENSION_FISICA = 100;

//otras funciones del programa

int main() {
    int arreglo[DIMENSION_FISICA];
    int dimension_logica = 0;
    return 0;
}
```

Si se hace necesario alterar el código para cambiar la dimensión física, basta con cambiar sólo una línea: la declaración de constante.



# Arreglos estáticos: pasaje por parámetro

Debido a que el pasaje de parámetros por valor implica una copia completa de la variable en la memoria, C++ **no permite el pasaje de arreglos por valor** y fuerza el pasaje por referencia. Esto implica que no es necesario incluir el operador & en un arreglo que aparece en la lista de parámetros de una función (automáticamente se pasará por referencia):

```
void funcion (int arreglo[], int dl) {  
    //cuerpo de la función  
}  
  
int main() {  
    int arreglo[50];  
    int dimension_logica = 0;  
    funcion(arreglo, dimension_logica);  
    return 0;  
}
```

Aunque no esté el operador  
&, el pasaje del arreglo se  
está haciendo por referencia

Llamada a la función, con el  
arreglo y la dimensión lógica  
como argumentos

# Arreglos estáticos: pasaje por parámetro

Lo que en verdad sucede es que, al pasar un arreglo como parámetro, el compilador lo reemplaza por una **referencia** al primer elemento. Los elementos siguientes se obtienen mediante un desplazamiento por la memoria, ya que se sabe de antemano que un arreglo se almacena en posiciones **contiguas**.

Debido a que los arreglos se pasan siempre por referencia, C++ no permite que una función retorne un arreglo estático.

Error

```
arreglo[] funcion (int arreglo[], int dl) {  
    //cuerpo de la función  
}
```

# Arreglos estáticos: iteración

Para acceder a todos los elementos de un arreglo (por ejemplo: para imprimirlos o para buscar la posición de un elemento en particular), iteraremos **incrementando** una variable numérica, que nos servirá de **índice**.

Pero ¿hasta dónde vamos a iterar?

La iteración termina cuando no hay más datos “útiles” en el arreglo. Esto es, hasta el elemento ubicado en la **dimensión lógica-1** (ya que el índice comienza en 0).

```
for (int i = 0; i < dimension_logica; i++)  
    cout << arreglo[i] << endl;
```

Este bucle imprime los elementos útiles del arreglo

Es importante tener en cuenta que nuestros datos siempre estarán almacenados de forma contigua en el arreglo, sin "huecos" en medio.

# Arreglos estáticos: inserción

Al insertar un nuevo elemento en un arreglo, la dimensión lógica nos indica hasta dónde "está lleno" el arreglo.

```
int numero;
cout << "Numero: (0 para cortar) ";
cin >> numero;
while (numero != 0 and dl < DIMENSION_FISICA) {
    arreglo[dl] = numero;
    dl++;
    cout << "Numero: (0 para cortar) ";
    cin >> numero;
}
```

Necesitamos una **doble condición de corte**: una para permitir al usuario finalizar la carga cuando quiera y otra para controlar que aún haya lugar para nuevos elementos.

Cuando la dimensión lógica es igual que la física, ya no hay espacio para insertar nuevos elementos.

Debemos permitir al usuario finalizar la carga cuando desee, siempre que haya espacio (de no ser así, siempre estaríamos cargando el arreglo completo y la dimensión lógica no tendría sentido).

# Arreglos estáticos: inserción al final

Conviene crear una función dedicada a la carga de nuevos elementos en el arreglo.

```
const int DIMENSION_FISICA = 100;

int cargar(int arreglo[], int dl) {
    int numero;
    cout << "Numero: (0 para cortar) ";
    cin >> numero;
    while (numero != 0 and dl < DIMENSION_FISICA) {
        arreglo[dl] = numero;
        dl++;
        cout << "Numero: (0 para cortar) ";
        cin >> numero;
    }
    return dl;
}

int main() {
    int numeros[DIMENSION_FISICA];
    int dimension_logica = 0;
    dimension_logica = cargar(numeros, dimension_logica);
}
```

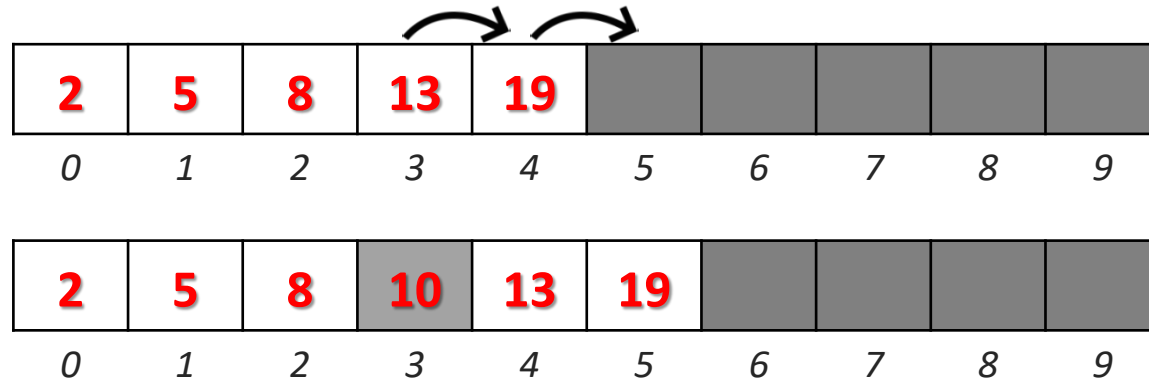
Almacenamos en una **variable aparte** el dato que usamos en la condición de corte. Luego de verificada la condición, **se inserta** en el arreglo.

Si almacenáramos directamente en el arreglo, como las condiciones de corte se verifican *después* de leer este dato, el arreglo podría estar lleno (d. lógica == d. física) y estaríamos guardando algo en una posición inválida (fuera del arreglo).

# Arreglos estáticos: inserción ordenada

Si queremos que un arreglo tenga sus elementos ordenados (de menor a mayor o viceversa), podemos insertar los elementos en orden cada vez que se agrega uno nuevo (también es posible ordenar el arreglo una vez que se han insertado todos, algo que no veremos en esta unidad).

Por ejemplo: si queremos insertar el número 10 en el arreglo graficado, que tiene números ordenados de menor a mayor, tendremos que "correr" el 13 y el 19 para "hacer lugar".



# Arreglos estáticos: inserción ordenada

Normalmente, este algoritmo tiene tres partes:

- 1- Buscar la posición donde debería insertarse el elemento.
- 2- Correr los elementos desde el final del arreglo hasta la posición donde se insertará.
- 3- Insertar el elemento e incrementar la dimensión lógica.

```
int i = 0;  
while (i < dl && arreglo[i] < nuevoDato)  
{  
    i++;  
}
```

Bucle para recorrer el arreglo  
buscando la posición

```
for (int j = dl; j > i; j--)  
{  
    arreglo[j] = arreglo[j-1];  
}
```

Bucle para correr todos los  
elementos necesarios

```
arreglo[i] = nuevoDato;  
dl++;
```

Se inserta el elemento en la  
posición encontrada



# Arreglos estáticos: inserción ordenada

```
const int DIMENSION_FISICA = 100;
int insertarOrdenado(int arreglo[], int dl, int nuevoDato) {
    int i = 0;
    while (i < dl && arreglo[i] < nuevoDato) {
        i++;
    }
    for (int j = dl; j > i; j--) {
        arreglo[j] = arreglo[j-1];
    }
    arreglo[i] = nuevoDato;
    return dl+1;
}

int cargar(int arreglo[], int dl) {
    int numero;
    cout << "Numero: (0 para cortar) ";
    cin >> numero;
    while (numero != 0 and dl < DIMENSION_FISICA) {
        dl = insertarOrdenado(arreglo, dl, numero);
        cout << "Numero: (0 para cortar) ";
        cin >> numero;
    }
    return dl;
}

int main() {
    int numeros[DIMENSION_FISICA];
    int dimension_logica = 0;
    dimension_logica = cargar(numeros, dimension_logica);
}
```

La solicitud de los datos la hacemos de igual manera que en la inserción al final, y sólo cambia la forma en que los datos se insertan en el arreglo.



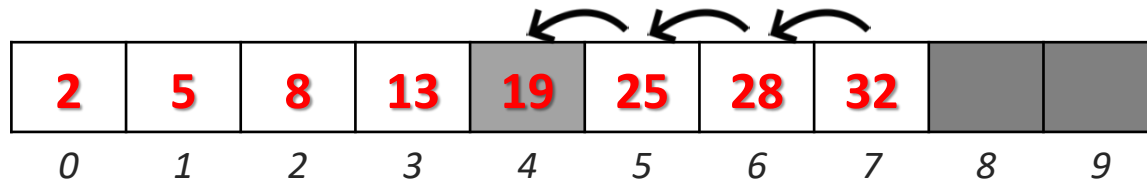
# Arreglos estáticos: eliminar un elemento

Los elementos de un arreglo no se “eliminan”, sino que se sobrescriben con otros datos.

Por ejemplo, si en un arreglo de dimensión lógica 8 queremos eliminar el elemento en la posición 4, haremos un **corrimiento** de los datos ubicados del 5 en adelante.

```
for (int i = 4; i < dimension_logica-1; i++) {  
    arreglo[i] = arreglo[i+1];  
}  
dimension_logica--;
```

Iteramos hasta el elemento ubicado en la posición dada por la dimensión lógica menos dos: el elemento en la posición 7 es el último, por lo que no se le debería colocar lo que hay en la posición siguiente (que no es válida)



Recordemos que no debemos dejar "huecos" de datos no útiles en el arreglo. Todos los datos que nos interesan deben estar contiguos, empezando desde la posición 0.

# Arreglos estáticos: eliminar un elemento

Normalmente, el algoritmo de eliminación tiene tres partes:

- 1- Buscar el elemento a eliminar (que también podría no existir dentro del arreglo), y obtener su posición en el índice.
- 2- Si se encontró, correr los elementos desde el siguiente a eliminar hasta el final (lógico) del arreglo, para sobrescribir al eliminado.
- 3- Decrementar la dimensión lógica.

```
int i = 0;
while (i < dimension_logica && arreglo[i] != elementoAEliminar)
{
    i++;
}
if (arreglo[i] == elementoAEliminar)
{
    for (int j = i; j < dimension_logica-1; j++)
    {
        arreglo[j] = arreglo[j+1];
    }
    dimension_logica--;
}
```

Bucle para recorrer el arreglo buscando el elemento

Bucle para correr todos los elementos necesarios

De esta forma, sólo se elimina la primera ocurrencia del elemento.

# Arreglos estáticos: eliminar ocurrencias

También podemos usar un algoritmo similar si queremos eliminar **todas** las ocurrencias de un elemento (y no sólo la primera).

```
int i=0;
while (i < dl) {
    if (arreglo[i] == elementoAEliminar) {
        for (int j = i; j < dl-1; j++) {
            arreglo[j] = arreglo[j+1];
        }
        dl--;
    }
    else
        i++;
}
```

Utilizamos while porque sólo queremos incrementar el índice si el elemento no se encontró en la posición actual

Si se encuentra una ocurrencia, realizamos el corrimiento visto anteriormente y decrementamos la dimensión lógica

Si la posición actual no contiene una ocurrencia del elemento a eliminar, avanzamos el índice para seguir buscando

# Estructuras

## struct

# Structs

- ▶ El programador puede definir un nuevo tipo en el que se asocian diferentes datos que tienen valores lógicamente relacionados y asociados bajo un único nombre. En C++ esto se logra con las structs.
- ▶ Es útil cuando se requiere agrupar varios datos que tienen sentido juntos (por ejemplo, la información personal de un empleado de una empresa).
- ▶ Una vez creado el tipo de dato, se podrán declarar variables de ese tipo. Incluso, pueden guardarse datos de este nuevo tipo dentro de contenedores (por ejemplo, dentro de un arreglo).

# Struct

Las **struct** son llamadas también **registros** o **record**.

Tienen muchos aspectos en común con los registros usados en bases de datos.

Nos van a permitir representar elementos del mundo real, que, en general, son más complejos que –por ejemplo– un número entero

Las **struct** permiten agrupar varios datos que mantengan algún tipo de **relación lógica**.

Los datos agrupados con **struct** pueden ser de distinto tipo: **es una estructura heterogénea**.

# Registros

Un registro es una colección de valores.

Los valores pueden ser de diferentes tipos →  
estructura de datos **heterogénea**.


El espacio de memoria ocupado por un registro es fijo →  
estructura **estática**.

El **acceso** a sus campos es **directo**.

# Definición

Un registro **se define** indicando la palabra reservada **struct**, el nombre de la estructura, y el tipo y nombre de los **campos** que componen la estructura.


```
struct identificador
{
    tipo campo1;
    tipo campo2;
    ...
}
```



no existe restricción para el tipo de los campos

Ejemplo:

```
struct Persona
{
    string nombre;
    int dni;
    string direccion;
};
```



campos de la estructura



# Campos

Cada **campo** puede tener un tipo diferente, que puede ser cualquier tipo de dato existente (int, string, bool...), incluso otras structs, contenedores, etc.

Una vez definida una struct, hemos creado un **nuevo tipo de dato**, que nos permitirá declarar variables de ese tipo.

En una variable de un tipo definido mediante struct, podemos acceder a cualquiera de sus campos sin necesidad de recorrer el resto de ellos (el acceso es **directo**).

Cada campo puede tratarse **individualmente** como un dato del tipo que le corresponda (por ejemplo, un campo de tipo int podría utilizarse dentro de una operación matemática).

# Ejemplos

Representar los datos de **un perro**.

Debería registrarse: nombre, raza, peso, color, edad, etc.



Representar los datos de **una elección electoral**. Debería registrarse: el distrito electoral, cantidad de votos para cada una de las listas presentadas, etc.



# Registros. Structs

¡Tomemos el ejemplo del perro!

Una manera natural y lógica de agrupar la información de cada perro en una sola estructura es declarar un tipo **struct** asociado a todos los datos que nos interesa guardar de cada perro.

Cada dato que compone la estructura es un **campo** de la **struct**.

Podemos definir al tipo **struct** del perro como:

Campos  
del  
registro  
Perro



- ✓ Nombre
- ✓ Raza
- ✓ Peso
- ✓ Color
- ✓ Edad



Los datos que almacenaremos dependerán del problema.

# Declaración de la struct y variables

Ejemplo de struct para almacenar datos de personas:

```
struct Persona
{
    string nombre;
    int dni;
    string direccion;
};
```

Ahora se pueden declarar variables del nuevo tipo creado:

```
Persona una_persona;
Persona otra_persona;
```

Para referenciar a cada campo de la estructura se utiliza el operador de selección (.) que se coloca entre el **nombre de la variable** y el **nombre del campo**:

```
una_persona.dni = 29568425;
```

# Declaración de la struct y variables

Una struct puede declararse en diferentes ámbitos. Si necesitamos que el tipo de dato que estamos definiendo exista en todo el programa, lo correcto será definirlo en el ámbito global.

```
struct Persona {  
    string nombre;  
    int dni;  
    string direccion;  
};  
  
void funcion(Persona dato) {  
    //cuerpo de la función  
}  
  
int main() {  
    Persona una_persona;  
    //resto de la función main  
}
```

ámbito global

# Declaración de la struct y variables

Al igual que con cualquier otro tipo de variables, se considera una mala práctica declarar variables de un tipo de struct en el ámbito global.

```
struct Persona {  
    string nombre;  
    int dni;  
    string direccion;  
};
```

ámbito global

```
Persona una_persona;
```

```
void funcion() {  
    //cuerpo de la función  
}
```

```
int main() {  
    //cuerpo de la función main  
}
```

Debe evitarse declarar variables de cualquier tipo en este ámbito. Lo correcto es declararlas en el ámbito local a la función o bloque donde vayan a utilizarse, pasándose como parámetro a otras funciones de ser necesario.

# Asignación

Campo a campo:



```
una_persona.direccion = "Gral. Paz 179";  
una_persona.dni = 29568425;  
una_persona.nombre = "Maria";
```

En una misma línea:



```
una_persona = {"Claudio", 53453455, "Alsina 43"};
```

el orden de los datos debe ser el mismo que en la declaración de la struct.

De una struct a otra:



```
otra_persona = una_persona;
```

se copian todos los campos de una variable a otra.

(suponemos que existen las variables una\_persona y otra\_persona, de tipo Persona)

# Asignación

Debemos recordar que el nombre de la struct creada es ahora el nombre de un tipo de dato, **no de una variable**.

Entonces, no es posible seleccionar un campo utilizando el nombre de la struct:

```
Persona.dni = 29568425;
```

## Error.

Persona es el nombre un tipo de dato, no el nombre de una variable.



# Comparación

Las operaciones de **comparación** no están definidas para la estructura completa. Pero sí se pueden comparar **los campos** de una struct:

```
if (una_persona.dni < otra_persona.dni)
    //bloque if
```

```
if (una_persona == otra_persona)
    //bloque if
```

Arroja un error indicando que la operación == no puede aplicarse entre dos variables de tipo Persona

# Impresión

De igual manera, no es posible imprimir los datos de una struct completa, sino que deben individualizarse los campos a mostrar:

```
cout << una_persona.nombre;  
cout << una_persona.dni;  
cout << una_persona.direccion;
```

```
cout << una_persona;
```



Error

# Operaciones con los campos

Cada campo de una struct tiene un tipo de dato asociado. Sobre cada campo podrán aplicarse las **operaciones definidas** para el **tipo de dato** del mismo.

<code>una_persona.nombre</code>	→	dato de tipo string
<code>una_persona.dni</code>	→	dato de tipo int
<code>una_persona.direccion</code>	→	dato de tipo string

Entonces, el campo de una struct puede usarse como cualquier otro dato de su tipo:

```
int cantidadDigitos(int numero) {  
    int cantidad = 0;  
    while (numero != 0) {  
        cantidad++;  
        numero /= 10;  
    }  
    return cantidad;  
}  
  
int main() {  
    Persona una_persona = { "Claudio", 53453455, "Alsina 43" };  
    cout << "Cantidad de dígitos del DNI: " << cantidadDigitos(una_persona.dni);  
}
```

una\_persona.dni es un dato de tipo int

# Arreglos con elementos de tipo struct

# Arreglos de structs: Declaración, asignación

Mientras todos los elementos de un **arreglo estático** sean del mismo tipo (homogéneo), ese tipo puede ser cualquiera, incluidos los definidos por el programador. Así, se puede declarar un arreglo cuyos elementos sean **structs**.

Para referenciar a un campo determinado de una struct almacenada en un arreglo, se debe acceder al elemento correspondiente (mediante su índice) y utilizar el punto para individualizar el campo:

```
Persona alumnos[5000];  
int dimL = 0;  
alumnos[0].direccion = "Gral Paz 179";  
alumnos[0].dni = 29568425;  
alumnos[0].nombre = "Maria";
```

**alumnos[0]** es un dato de tipo Persona, al igual que en la declaración:  
**Persona una\_persona;**

# Arreglos de structs: Elementos y campos

En un arreglo de structs, cada elemento es una struct. Por ende, si se accede a un **elemento del arreglo**, se estará obteniendo una **struct**. Y, obtenida la struct, se puede acceder a sus campos.

`alumnos[0]`



Se obtiene una struct de tipo Persona (la que esté almacenada en la posición 0 del arreglo).

`alumnos[0].dni`



Se obtiene un número de tipo int (el que esté almacenado en el campo dni de la struct en la posición 0 del arreglo).

Con cada uno se pueden hacer las operaciones que el tipo permita. Por ejemplo, **no podría asignarse un dato de tipo int a una struct de tipo Persona**, de la misma manera que no puede almacenarse un dato de tipo float en una variable de tipo string.

`alumnos[0] = 25;`



Error

# Arreglos de structs: Elementos y campos

Lo importante es, una vez obtenido un dato, saber con qué tipo de dato estamos trabajando y qué cosas es posible hacer con ese tipo. Ya no importa si el dato está dentro de un arreglo, si es parte de una struct o dónde está almacenado.

Si un elemento de un arreglo es de tipo Persona, podremos hacer con ese dato todo lo que se pueda hacer con un dato de tipo Persona. Si un campo de una struct es de tipo string, podremos hacer con ese dato todo lo que se pueda hacer con un dato de tipo string.

# Arreglos de structs: Elementos y campos

Podría pasarse una struct contenida en el arreglo como **argumento** en una llamada a una función que reciba una struct del mismo tipo (en este ejemplo, de tipo Persona):

```
int digitosDNI(Persona alumno) {  
    int digitos = 0;  
    while (alumno.dni != 0) {  
        digitos++;  
        alumno.dni = alumno.dni/10;  
    }  
    return digitos;  
}  
  
int main() {  
    Persona alumnos[5000];  
    int dimL = 0;  
    dimL = cargarAlumnos(alumnos, dimL);  
    cout << "Cantidad de dígitos del DNI: " << digitosDNI(alumnos[3]);  
}
```

Recordemos que, como el parámetro de tipo Persona está pasado por valor, el dato original dentro del arreglo no se modifica.

alumnos[3] es un dato de tipo Persona



# Arreglos de structs: Elementos y campos

Debido a que cada campo puede accederse de forma individual, también podría pasarse únicamente uno de los campos de la struct contenida en el arreglo (y en este caso sería lo óptimo, ya que la función sólo trabaja con un número):

```
int cantDigitos(int n) {  
    int digitos = 0;  
    while (n != 0) {  
        n = n/10;  
        digitos++;  
    }  
    return digitos;  
}  
  
int main() {  
    Persona alumnos[100];  
    int dimL = 0;  
    dimL = cargarAlumnos(alumnos, dimL);  
    cout << "Cantidad de dígitos del DNI: " << cantDigitos(alumnos[3].dni);  
}
```

alumnos[3].dni es un dato de tipo int

Esto ya lo vimos, cuando aprendimos sobre "operaciones con los campos" de una struct.

# Arreglos de structs: Impresión

Como ya hemos visto, no es posible imprimir una struct completa. Entonces, si queremos imprimir las structs contenidas en el arreglo, deberemos iterar por él, imprimiendo campo a campo:

```
void imprimir(Persona arreglo[], int dl) {  
    for (int i = 0; i != dl; i++) {  
        cout << arreglo[i].nombre << endl;  
        cout << arreglo[i].dni << endl;  
        cout << arreglo[i].direccion << endl;  
    }  
}
```

# Arreglos de structs: Carga de datos

Los algoritmos de carga son los mismos que usamos para cargar cualquier arreglo, sólo que ahora indicamos cada campo de la struct donde queremos almacenar algo. Por ejemplo, para insertar al final:

```
int cargar(Persona arreglo[], int dl) {  
    Persona p;  
    cout << "Nombre: (x para cortar) ";  
    getline(cin>>ws, p.nombre);  
    while (p.nombre != "x" and dl < DIMENSION_FISICA) {  
        cout << "DNI: ";  
        cin >> p.dni;  
        cout << "Dirección: ";  
        getline(cin>>ws, p.direccion);  
        arreglo[dl] = p;  
        dl++;  
        cout << "Nombre: (x para cortar) ";  
        getline(cin>>ws, p.nombre);  
    }  
    return dl;  
}
```

Recordemos que no cargamos los datos directamente en el arreglo sino que usamos una variable aparte, para evitar escribir más allá del final del arreglo si es que está lleno.

Como p es de tipo Persona y el arreglo contiene elementos de tipo Persona, podemos asignar p a una posición del arreglo.

# FIN UNIDAD 2

Esta presentación fue diseñada por el siguiente equipo docente:

Dra. Claudia Russo  
Lic. Paula Lencina  
Lic. Cecilia Rastelli  
AC María Lanzillotta  
AC. Marina Rodríguez  
AC. David Fernández  
Prog. Trinidad Picco



**Atribución - No Comercial** (*by-nc*): Se permite la generación de obras derivadas siempre que no se haga con fines comerciales. Tampoco se puede utilizar la obra original con fines comerciales. Esta licencia no es una licencia libre.