



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 1:

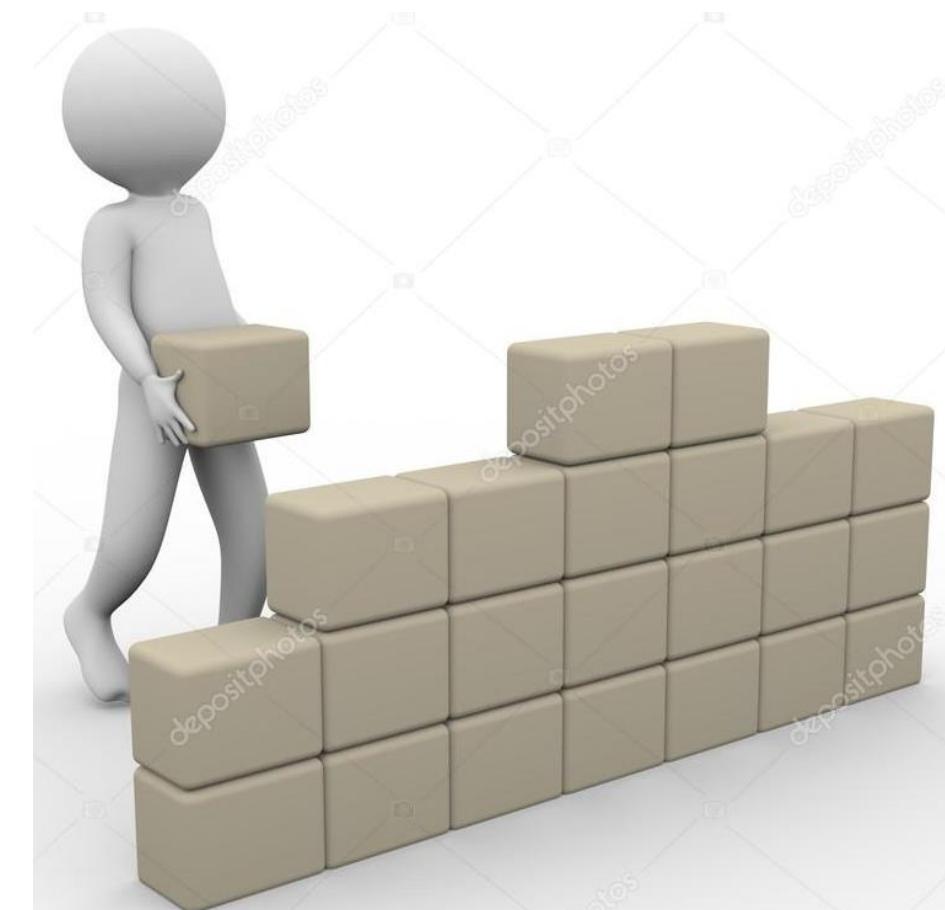
Recursividad e Iteración

Contenido

1. Funciones en C++: Definición
2. Declaración (prototipo) vs definición
3. Llamada de una función
4. Función main. La pila de ejecución
5. Paso de parámetros por valor y por referencia
6. Paso de parámetros de tipo arreglos
7. Variables y alcance en funciones
8. Sobrecarga y Plantilla de funciones
9. Recursividad

Introducción

- Un programa de 1000 líneas dentro de main() es difícil de leer, mantener y escalar. En proyectos reales (como software de ingeniería, videojuegos, sistemas embebidos, etc.) es insostenible.
- Las funciones permiten:
 - ✓ Dividir el trabajo en tareas pequeñas y manejables.
 - ✓ Reutilizar código.
 - ✓ Organizar el código en bloques con responsabilidad única.
 - ✓ Hacer pruebas más fáciles y más precisas (unit testing).
 - ✓ Fomentar trabajo colaborativo



1. Funciones en C++: Definición

- Una función en C++ es un bloque de código con nombre que:
 - ✓ Puede tomar parámetros (entradas)
 - ✓ Realiza un proceso
 - ✓ Puede devolver un resultado (salida)

- **Sintaxis básica**

```
tipo_retorno nombre_funcion(tipo_1 arg_1, ..., tipo_n arg_n) {  
    // cuerpo  
    return valor_retorno; // opcional  
}
```

- **Ejemplo:**

```
double areaRectangulo(double base, double altura) {  
    return base * altura;  
}
```

2. Declaración (prototipo) vs definición

- **Declaración o prototipo.** Se escribe antes de main() para decirle al compilador que la función existe.

```
double areaRectangulo(double base, double altura);
```

- ✓ Permite organizar el código por secciones (main arriba, funciones abajo).
- ✓ Mejora la legibilidad y el orden.
- ✓ Necesario cuando la función es definida después de ser llamada.
- ✓ `areaRectangulo(double, double);` firma de la función (util en overloading)

- **Definición.** Es la implementación completa

```
double areaRectangulo(double base, double altura) {  
    return base * altura;  
}
```

3. Llamada de una función

- Cuando una función es invocada:

- ✓ Se evalúan los argumentos que se pasan.
- ✓ El control del programa se transfiere a la función.
- ✓ Se ejecuta el cuerpo de la función.
- ✓ Se ejecuta una instrucción return (explícita o implícita).
- ✓ El control retorna al punto de llamada en el program

Las funciones pueden ser llamadas dentro de otras funciones o incluso dentro de expresiones.

```
int sumar(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    int x = 5;  
    int y = 10;  
    int resultado = sumar(x, y); // ← llamada  
    cout << "Resultado: " << resultado << endl;  
    return 0;  
}
```

- main() se ejecuta.
- sumar(x, y) es llamada → a = 5, b = 10.
- Se ejecuta return a + b → devuelve 15.
- Ese valor es asignado a resultado.
- cout imprime el resultado.

4. main() como función especial

- main() es una función especial: es el punto de entrada del programa.

```
int main() {  
    // ...  
    return 0;  
}
```

```
int main(int argc, char* argv[]) {  
    //para argumentos desde la línea de comandos  
}
```

- **La pila de ejecución (call stack):** Cada vez que se llama una función, se crea un marco (**frame**) en la pila. Cuando la función termina, ese marco se destruye y se retorna al anterior.

```
int doble(int x) {  
    return x * 2;  
}
```

```
int sumarDobles(int a, int b) {  
    return doble(a) + doble(b);  
}
```

```
int main() {  
    cout << sumarDobles(3, 4) << endl;  
    return 0;  
}
```

Stack de llamadas (flujo):

1. main() llama sumarDobles(3, 4)
2. sumarDobles llama doble(3)
3. doble(3) devuelve 6
4. sumarDobles llama doble(4)
5. doble(4) devuelve 8
6. sumarDobles devuelve 14 a main()

5. Paso de parámetro por valor y por fererencia

¿Qué es el paso de parámetros?

Cuando se llama a una función con argumentos, esos valores deben ser recibidos por parámetros. Hay dos formas comunes de hacerlo en C++:

Forma	¿Copia datos?	¿Puede modificar los originales?
Por valor	SI	NO
Por referencia	NO	SI

- **Paso por Valor** (`int x`) Se pasa una copia del valor. Lo que ocurra dentro de la función no afecta al original.
- **Paso por Referencia** (`int& x`) Se pasa una referencia al valor original. Lo que ocurra dentro de la función afecta al original.
- **Y si no quieres modificar, pero pasas por referencia?** Puedes usar `const` para seguridad contra modificaciones (`const int& x`)

Ejemplo: Paso por valor (duplicar el valor de un entero)

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

void duplicarValor(int x) {
    x = x * 2;
    cout << "Dentro de la función: " << x << endl;
}

int main() {
    int numero = 5;
    cout << "Antes de llamar a la función: " << numero << endl;
    duplicarValor(numero);
    cout << "Después de llamar a la función: " << numero << endl;
    return 0;
}
```

Antes de llamar a la función: 5
Dentro de la función: 10
Después de llamar a la función: 5

Ejemplo: Paso por referencia (duplicar el valor de un entero)

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

```
void duplicarValor(int& x) {
```

```
    x = x * 2;
```

```
    cout << "Dentro de la función: " << x << endl;
```

```
}
```

```
int main() {
```

```
    int numero = 5;
```

```
    cout << "Antes de llamar a la función: " << numero << endl;
```

```
    duplicarValor(numero);
```

```
    cout << "Después de llamar a la función: " << numero << endl;
```

```
    return 0;
```

```
}
```

```
Antes de llamar a la función: 5
```

```
Dentro de la función: 10
```

```
Después de llamar a la función: 10
```

6. Parámetros tipo arreglos.

- En C++, cuando pasamos un arreglo a una función:
No se copia el arreglo → se pasa la dirección del primer elemento (puntero ??)
(los arreglos de pasan por referencia automáticamente)
Por lo tanto, la función puede modificar los elementos originales del arreglo.
Se pierde el tamaño del arreglo, a menos que lo pases como parámetro adicional.
- .
- **Prototipo típico:**
`void mostrar(int arr[], int tam);`
- **Alternativa moderna: std::vector**
 - ✓ Tiene tamaño interno (.size())
 - ✓ Paso flexible, por valor o referencia, pero más seguro
 - ✓ Más flexible (puede crecer dinámicamente)

Ejemplo: duplicar los elementos de un arreglo

```
#include <iostream>
using namespace std;
// Definición de la función
void modificarArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {arr[i] *= 2;}
}
int main() {
    int miArray[] = {1, 2, 3, 4, 5};
    int tamano = sizeof(miArray) / sizeof(miArray[0]);
    modificarArray(miArray, tamano); // llamada

    cout << "Array modificado: ";
    for (int i = 0; i < tamano; i++) {
        cout << miArray[i] << " ";
    }
    cout << endl;
}
```

Array modificado: 2 4 6 8 10

Parámetros tipo matriz

- Se **debe especificar el tamaño de las columnas**. El compilador necesita saber cuántas columnas hay para poder hacer el cálculo de desplazamiento correcto: $\text{mat}[i][j] \rightarrow *(\text{mat} + i \times \text{columnas} + j)$

```
void imprimirMatriz(int matriz[][][4]);
```

- En la **llamada** solo se especifica el nombre de la matriz

```
imprimirMatriz(miMatriz);
```

Ejemplo: imprimir una matriz

```
#include <iostream>
using namespace std;
const int FILAS = 3; //Variable global
const int COLUMNAS = 3; //Variable global

// Función para imprime una matriz
void imprimirMatriz(int matriz[][][COLUMNAS]) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            cout << matriz[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int miMatriz[FILAS][COLUMNAS] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    imprimirMatriz(miMatriz, FILAS); //llamada

    return 0;
}
```

1	2	3
4	5	6
7	8	9

7. Variables y alcance en funciones

- El alcance (scope) de una variable se refiere a dónde puede ser accedida o utilizada en el código. En C++, hay principalmente tres tipos de alcance:

Tipo de variable	¿Dónde vive?	Acceso
Local	Dentro de un bloque {}	Sólo ese bloque
Global	Fuera de cualquier función	Todo el archivo
Estática (static)	Dentro de la función	Sólo la función (mantiene su valor entre llamadas)

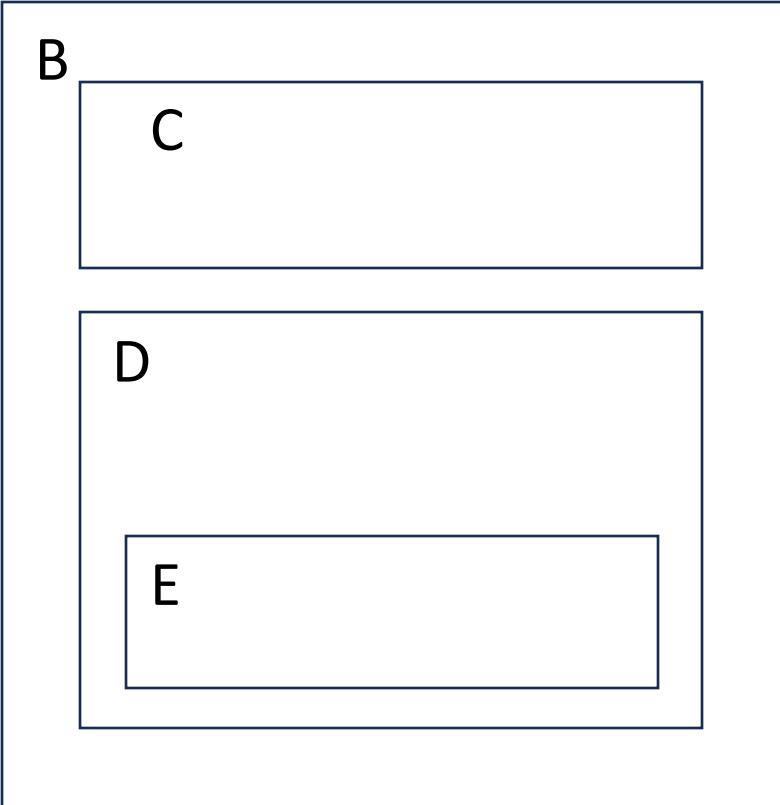
- **Variables Locales**: declarada dentro de una función o bloque, se crean al entrar en el bloque y se destruyen al salir.
- **Variables Globales**: declarada fuera de cualquier función. Se pueden usar en todo el archivo fuente.
- **Variable staticas**: conservan su valor entre llamadas

Tiempo de vida de las variables

- **Peligros de las variables globales:**
 - ✓ Difícil de rastrear errores.
 - ✓ Rompe el encapsulamiento.
 - ✓ Reduce la reutilización de funciones
- **Reglas de Alcance en C++:**
 - ✓ Una variable solo vive en el bloque donde fue declarada.
 - ✓ No se puede acceder a una variable antes de declararla.
 - ✓ Se pueden tener variables con el mismo nombre en bloques distintos.
- **Tiempo de vida de las variables:**
 - ✓ Las variables locales existen solo mientras el bloque se ejecuta.
 - ✓ Las globales existen durante toda la ejecución del programa.
 - ✓ Las static locales tienen alcance limitado, pero vida extendida

Identificando variables locales y globales

A



Variables definidas en:

A

B

C

D

E

Son accesibles desde:

A, B, C, D, E

B, C, D, E

C

D, E

E

8. Sobrecarga de funciones (overloading)

- Permite declarar varias funciones con el mismo nombre siempre que tengan diferentes parámetros (en cantidad o tipo) . Ejemplo:

```
int mul(int,int);
float mul(float,int);

int main(){
    int r1 = mul(2,3);
    float r2 = mul(0.2,4);
    return 0;
}

int mul(int a,int b){ return a*b; }
float mul(double x, int y){ return x*y; }
```

```
int sumar(int a, int b);
int sumar(int a, int b, int c);
int multiplicar(int a, int b);
int multiplicar(int a, int b, int c);

int main() {
    cout << sumar(5, 3) << endl;
    cout << sumar(5, 3, 2) << endl;
    cout << multiplicar(5, 3) << endl;
    cout << multiplicar(5, 3, 2) << endl;

    return 0;
}

// implemente las funciones aquí!!
```

int f(int x); double f(int x); define una sobrecarga?

Plantilla de Funciones (templates)

- C++ permite crear funciones genéricas, es decir, una función que puede tomar cualquier tipo de dato como argumento.

- **Sintaxis:**

```
template <typename T>
T duplicar(T valor) {
    return valor * 2;
}
```

- **Ejemplo:**

```
template <typename T>
void imprimirGenerico(T dato) {
    cout << "Datos: " << dato << endl;
}

imprimirGenerico(5);           // int
imprimirGenerico(3.14);        // double
imprimirGenerico("Hola");      // const char
```

9. Recursividad

- Es una técnica de programación en la que una función se llama a sí misma para resolver un problema, dividiéndolo en subproblemas más pequeños. Se compone de:
 - ✓ **Caso base:** define cuándo detenerse.
 - ✓ **Caso recursivo:** define cómo dividir el problema y llamarse a sí misma.
- **¿Como funciona la recursión?**
Cada llamada a la función se apila en la memoria (stack) y se resuelve cuando se alcanza el caso base.
- **Importancia del caso base:** Si no se define correctamente el caso base, se produce un ciclo infinito y eventualmente un desbordamiento de pila.

Ejemplo: Factorial recursivo

```
int factorialRecursivo(int n) {  
    if (n == 0)  
        return 1;  
    return n * factorialRecursivo(n - 1);  
}
```

¿Como se ejecuta la recursión?

Ejemplo para n = 3:

factorial(3)
→ 3 * factorial(2)
 → 2 * factorial(1)
 → 1 * factorial(0)
 → 1

Otros ejemplos comunes de recursividad:

1. Fibonacci
2. Potencias
3. Inversión de una cadena
4. Suma de elementos en un arreglo
5. mcm, mcd
6. Palabra palíndroma
7. Imprimir números en forma ascendente
8. Sumar los dígitos de un número
9. Contar los dígitos de un número
10. Convertir un número a base binaria

Ejemplo: Factorial con recursividad mutua

En una recursividad mutua (indirecta), una función llama a otra, que a su vez llama de regreso a la primera, creando un ciclo recursivo.

```
// Declaración anticipada
int factorialB(int n);

int factorialA(int n) {
    if (n <= 1)
        return 1;
    return n * factorialB(n - 1);
}

int factorialB(int n) {
    if (n <= 1)
        return 1;
    return n * factorialA(n - 1);
}
```

¿Como se ejecuta la recursión? Ejemplo para $n = 5$:
Llamando a $\text{factorialA}(5)$:

$\text{factorialA}(5)$
→ $5 * \text{factorialB}(4)$
→ $4 * \text{factorialA}(3)$
→ $3 * \text{factorialB}(2)$
→ $2 * \text{factorialA}(1)$
→ retorna 1 (caso base)

Recursión vs iteración

Característica	Recursividad	Iteración
Concepto	Función se llama a sí misma	Uso de bucles (for, while)
Uso de memoria	Más consumo (por el stack)	Menor consumo
Código	Más compacto	Más explícito
Velocidad	Generalmente más lenta	Más rápido
Casos ideales	Problemas de naturaleza recursiva	Procesos repetitivos simples

Ejemplo: Recursión vs iteración

```
int factorialIterativo(int n) {  
    int resultado = 1;  
    for (int i = 2; i <= n; ++i) {  
        resultado *= i;  
    }  
    return resultado;  
}
```

```
int factorialRecursivo(int n) {  
    if (n <= 1)  
        return 1;  
    return n * factorialRecursivo(n - 1);  
}
```

Ejemplo: Factorial con recursividad mutua

En una recursividad mutua (indirecta) Es, una función llama a otra, que a su vez llama de regreso a la primera, creando un ciclo recursivo.

```
// Declaración anticipada
int factorialB(int n);

int factorialA(int n) {
    if (n <= 1)
        return 1;
    return n * factorialB(n - 1);
}

int factorialB(int n) {
    if (n <= 1)
        return 1;
    return n * factorialA(n - 1);
}
```

¿Como se ejecuta la recursión? Ejemplo para n = 5:
Llamando a factorialA(5);

factorialA(5)
→ 5 * factorialB(4)
→ 4 * factorialA(3)
→ 3 * factorialB(2)
→ 2 * factorialA(1)
→ retorna 1 (caso base)



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**

- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 2:

Algoritmos de Ordenamiento y Búsqueda I

Contenido

1. Ordenamiento

- Algoritmo de burbuja
- Algoritmo de selección
- Algoritmo de inserción

2. Búsqueda

- Algoritmo de búsqueda lineal

1. Ordenamiento

- Es una operación fundamental en la computación. Permite organizar los datos, facilitando su acceso y procesamiento:
 - ✓ Un conjunto de datos ordenado permite aplicar algoritmos de búsqueda más eficientes (búsqueda binaria). Ejemplo: un directorio telefónico está ordenado alfabéticamente.
 - ✓ Los datos ordenados facilitan la interpretación y toma de decisiones. Ejemplo: En estadísticas, ordenar datos ayuda a calcular la mediana o el percentil.
 - ✓ Algoritmos de aprendizaje automático y minería de datos suelen requerir datos ordenados para mejorar la eficiencia y precisión de los modelos.
- **Ejercicio:** Investiga cómo los motores de búsqueda como Google utilizan algoritmos de ordenamiento para mostrar resultados relevantes en cuestión de milisegundos.

Convenciones

- En esta presentación asumiremos que los elementos de los arreglos son números enteros y se ordenará en forma ascendente (de menor a mayor).
- Para **intercambiar** dos elementos en un arreglo es necesario utilizar una variable temporal (temp) para guardar una copia del primer elemento, asignarle el segundo al primero y el temporal al segundo.

Ejercicio: implemente la función intercambio

- Si el arreglo de nombre lista tiene N elementos, el primer elemento es lista[0] y el último es lista[N-1].

Algoritmo de Burbuja (Bubble sort)

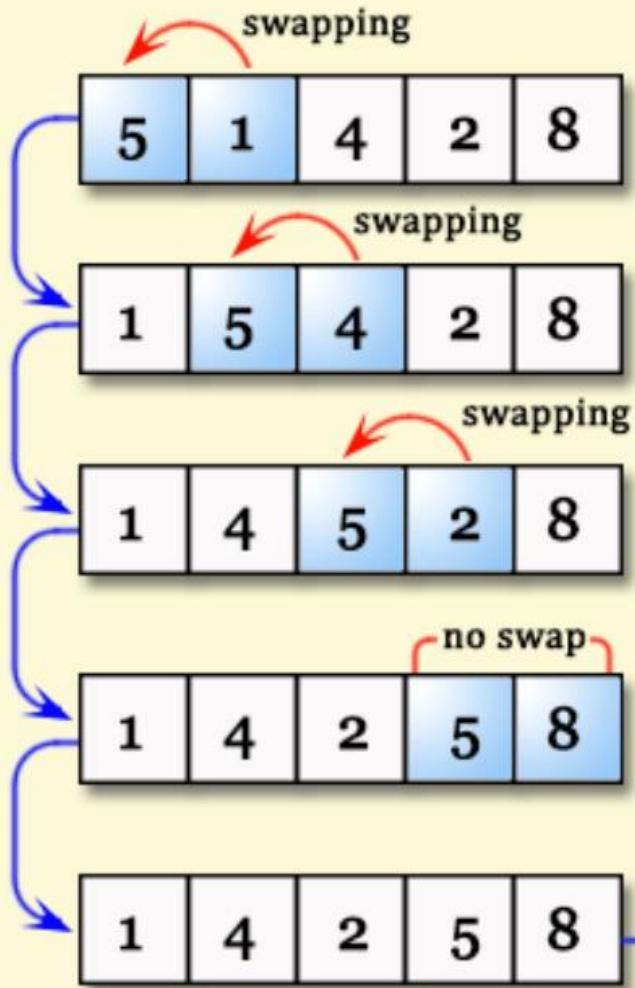
- Compara pares adyacentes y los intercambia si están en el orden incorrecto.
- Al final de cada iteración (pasada), el mayor elemento aparece al final del arreglo (“burbuja”). En general se requieren $N-1$ pasadas para un arreglo de tamaño N
- El algoritmo puede optimizarse teniendo en cuenta los intercambios. Si en una pasada no hay intercambio significa que el arreglo ya está ordenado.
 - ✓ Se suele utilizar una variable que indica si ocurre o no intercambios. En caso no ocurra intercambio se debe terminar la iteración.

Ejemplo: algoritmo de burbuja

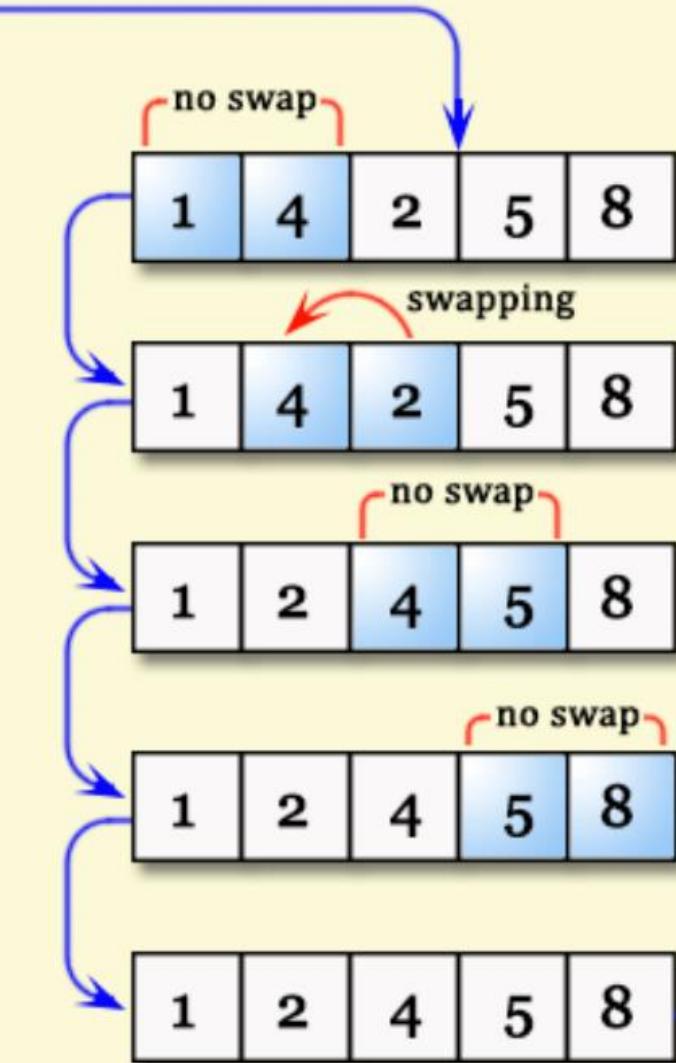
- Arreglo desordenado:

5	1	4	2	8
---	---	---	---	---

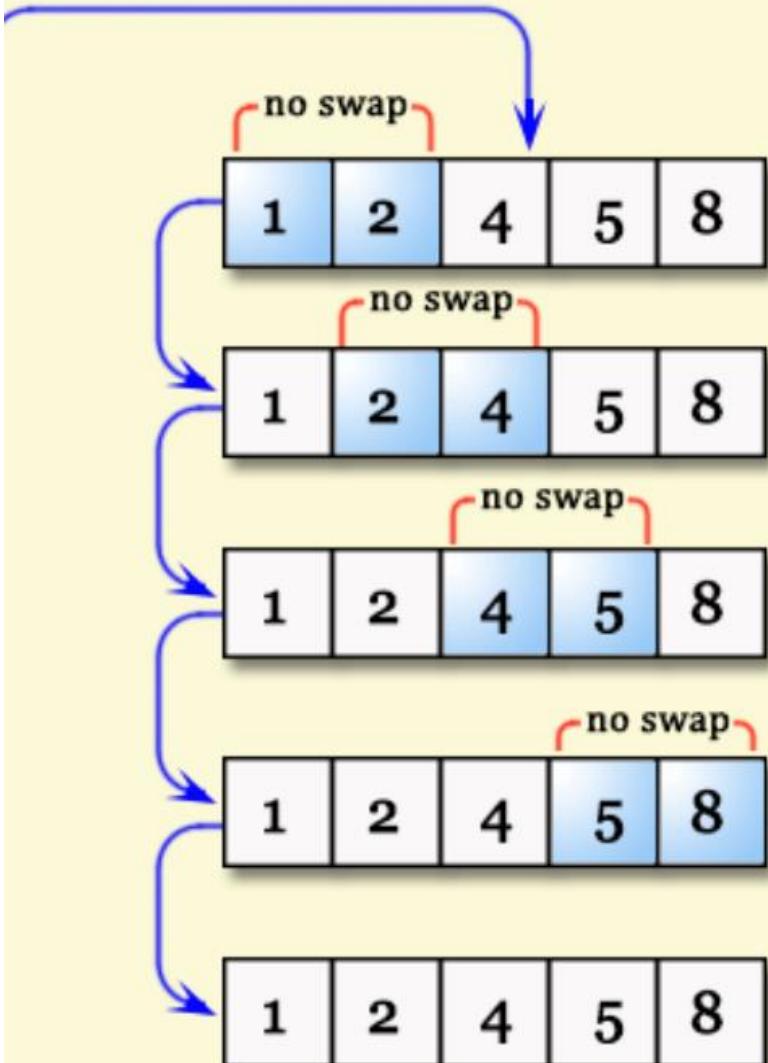
First Pass



Second Pass



Third Pass



Ejercicios

- Implementar el algoritmo de burbuja. Utilice funciones.
- Implementar una variante que detecte si el arreglo ya está ordenado y termine antes de tiempo.
- Escribir una versión utilizando recursividad.
- Escribir una versión para cadenas

```
void burbuja(int arr[], int n){
    for(int i = 0; i < n - 1; ++i){
        for(int j = 0; j < n - 1 - i; ++j){
            if(arr[j] > arr[j+1]){
                swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

Algoritmo de Selección (Selection Sort)

- Encuentra el menor elemento del arreglo y lo intercambia con el primer elemento
- Buscar el siguiente menor elemento en el resto del arreglo y lo intercambia con el segundo elemento.
- En general, en cada pasada selecciona el menor elemento entre una posición i y el final del arreglo e intercambia el mínimo con el elemento de la posición i .
- Si el arreglo tiene N elementos, el proceso se repite $N-1$ veces, ya que el último elemento no se compara.

Ejemplo: Algoritmo de Selección

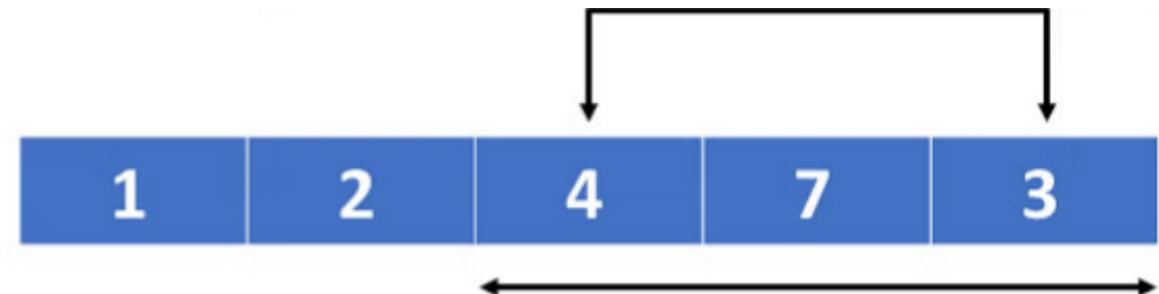
- Arreglo desordenado:

7	1	4	2	3
---	---	---	---	---

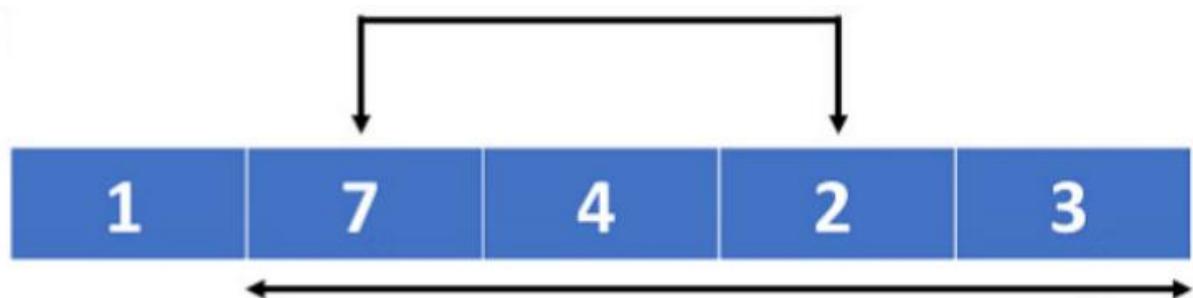
Pasada 1



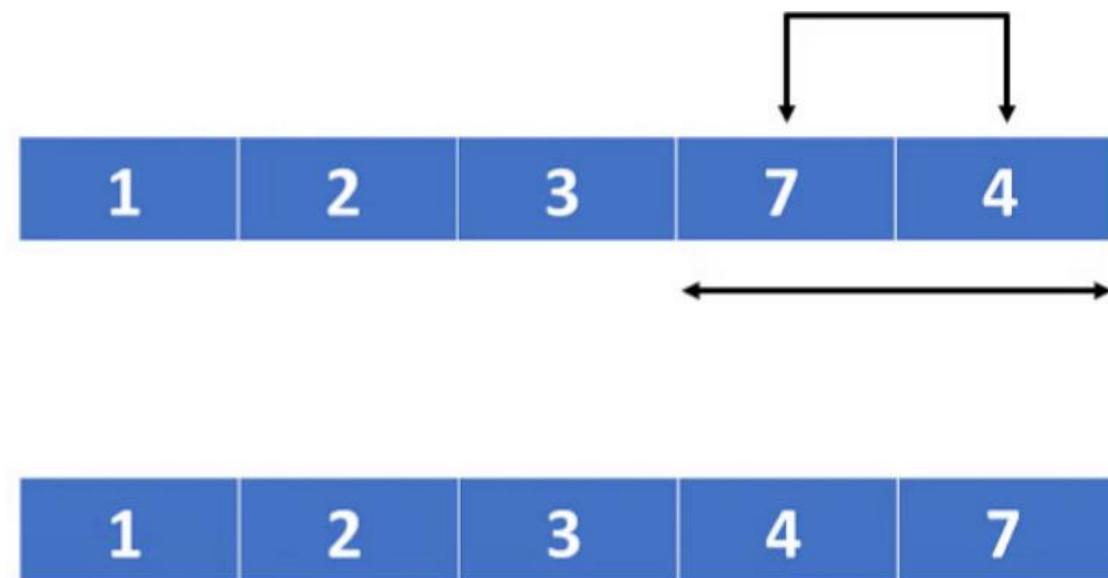
Pasada 3



Pasada 2



Pasada 4

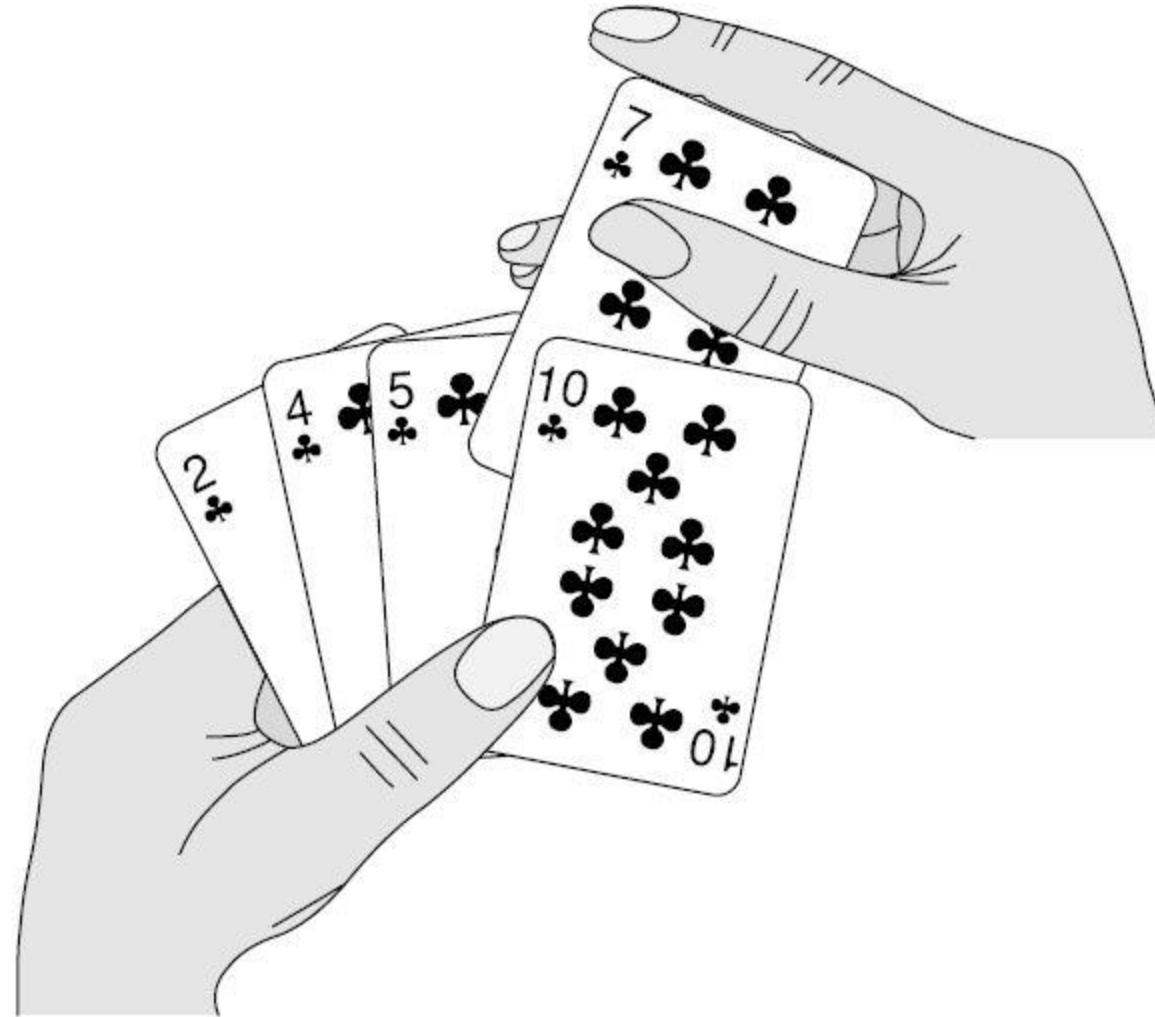


Ejercicios

- Escribir una función que implemente el algoritmo de selección
- Escribir una versión recursiva del algoritmo.
- Escribir una versión para cadenas.
- Verificar los cambios que deben realizarse para un ordenamiento decreciente.

```
void seleccion(int arr[], int n){
    for(int i = 0; i < n - 1; ++i){
        int indMin = i;
        for(int j = i + 1; j < n; ++j){
            if(arr[j] < arr[indMin]){
                indMin = j;
            }
        }
        swap(arr[i], arr[indMin]);
    }
}
```

Algoritmo de Inserción (Insertion Sort)



4. Algoritmo de Inserción (Insertion Sort)

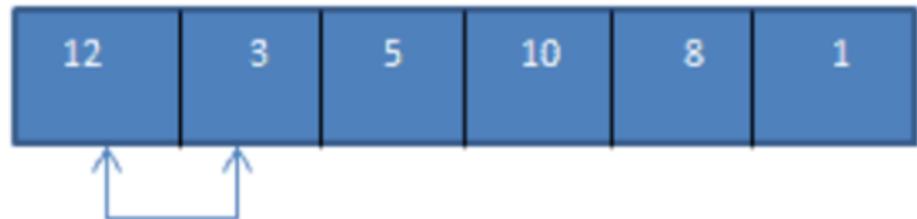
- El algoritmo se asemeja al hecho de insertar una carta en su posición correcta dentro de una lista que ya está ordenada.
- El primer elemento lista[0] se considera ordenado (una sola carta).
- Se inserta lista[1] en la posición correcta (delante o detrás de lista[0]) y así sucesivamente.
- En general, toma cada elemento y lo inserta en la posición correcta dentro de la parte ordenada.
- Para un arreglo de N elementos, deben hacerse N-1 pasadas, pues el primer elemento ya está ordenado.

Ejemplo: Algoritmo de Inserción

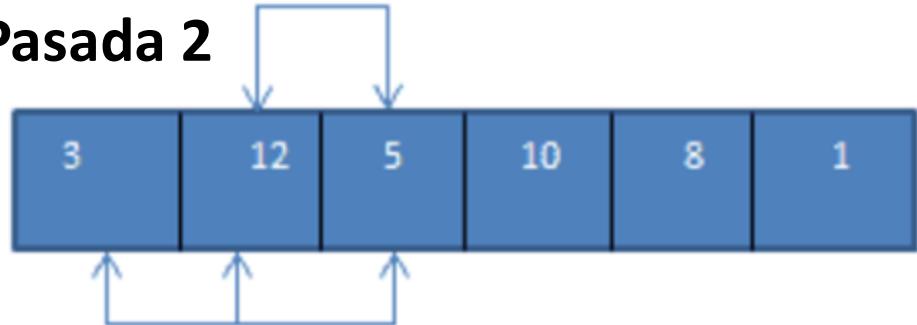
- Arreglo a desordenado:

12	3	5	10	8	1
----	---	---	----	---	---

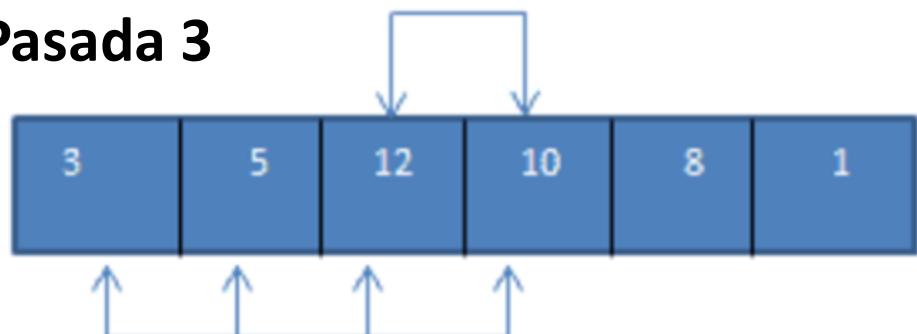
Pasada 1



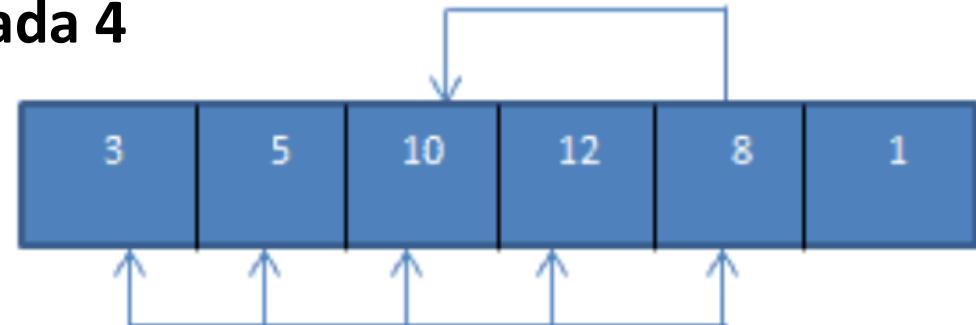
Pasada 2



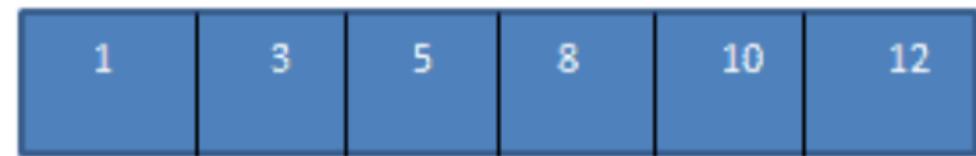
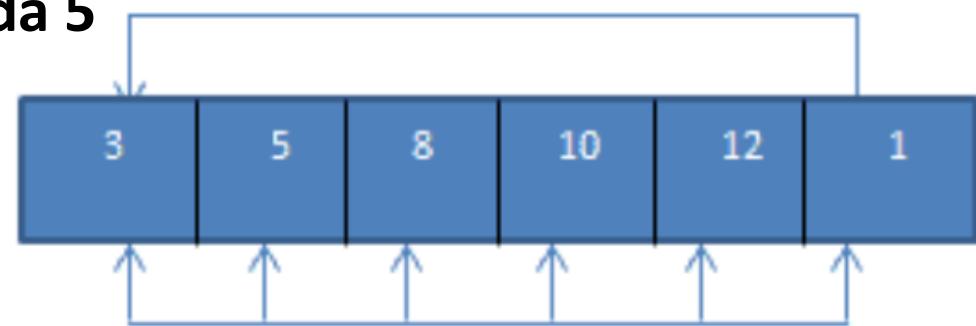
Pasada 3



Pasada 4



Pasada 5



Ejercicios

- Utilice funciones para implementar el algoritmo de inserción
- Escribir una versión para cadenas.
- Verificar los cambios que deben hacerse para un ordenamiento decreciente.
- Analice una versión recursiva.

```
void insercion(int arr[], int n){
    for(int i = 1; i < n ; ++i){
        int temp = arr[i];
        int j = i-1;
        while(j>=0 && arr[j] > temp){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}
```

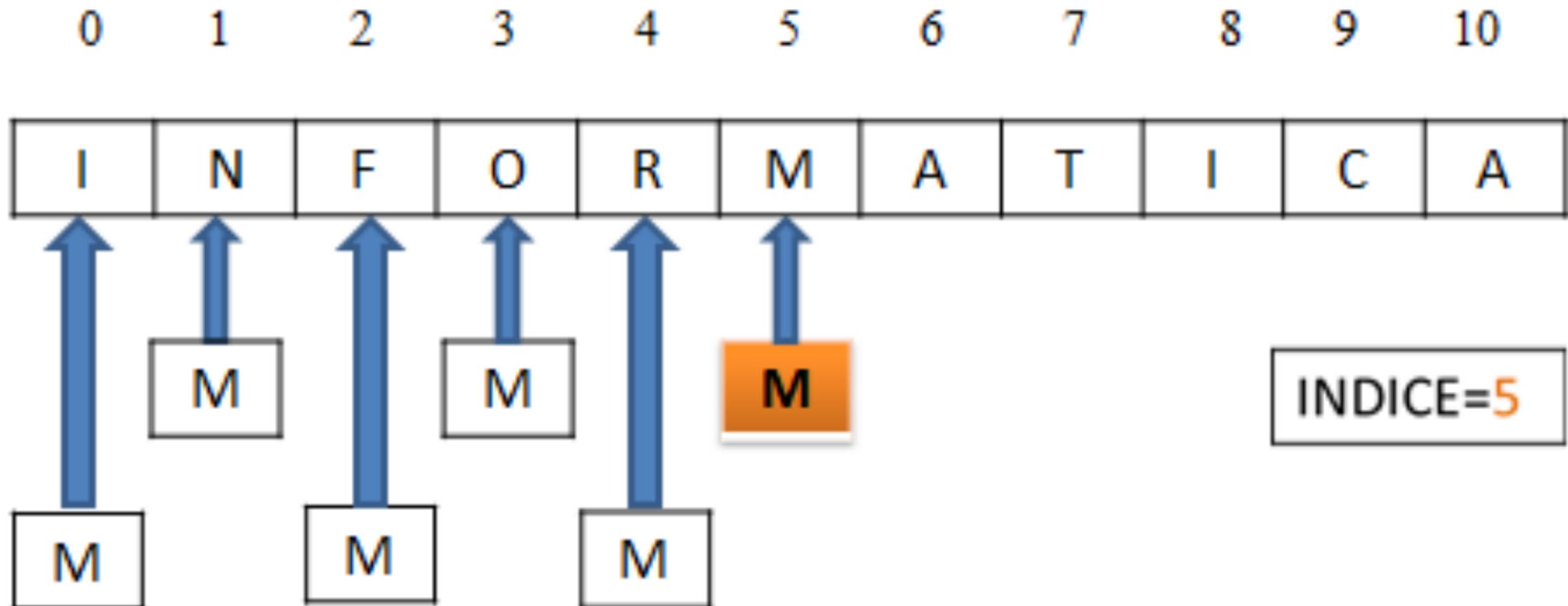
2. Búsqueda

- Es una de las operaciones más frecuentes en informática:
 - ✓ Permite recuperar datos de grandes volúmenes de información de manera eficiente.
 - ✓ Las aplicaciones requieren respuestas rápidas para búsquedas en bases de datos y archivos. Ejemplo: App de una tienda en línea, bibliotecas digitales.
 - ✓ Se usa en detección de patrones, reconocimiento de imágenes y en sistemas de seguridad para verificar identidades o detectar amenazas.
- **Ejercicio:** Investigar problemas reales en el que una búsqueda rápida sea crucial. ¿Cómo mejorarías la eficiencia de la búsqueda en ese caso?

Algoritmo de Búsqueda Lineal

- Recorre el arreglo comparando uno a uno cada elemento hasta encontrar el valor buscado.
- Se comporta bien con arreglos pequeños o desordenados.

Búsqueda lineal



Ejercicios

- Implemente el algoritmo de búsqueda lineal utilizando funciones
- Implementar una versión recursiva del algoritmo de búsqueda lineal.
- Modificar la función para contar cuántas veces aparece un número en un arreglo.
- Escribir una versión para cadenas.
- Implementar un programa que permita encontrar la última ocurrencia de un determinado carácter en una cadena

```
int busquedaLineal(int arr[], int n, int valor){
    for(int i = 0; i < n; ++i){
        if(valor == arr[i]){
            return i;
        }
    }
    return -1;
}
```

Ejercicios

- ¿ Cómo generar números aleatorios flotantes en el intervalo $[0, 1]$? ¿En general en un intervalo la forma $[a, b]$? .
Genere arreglos con elementos aleatorios del tipo double.
- Realice la implementación modular de los algoritmos de ordenamiento y búsqueda que veremos en el curso (debe dividir su código en funciones bien definidas y distribuirlo en varios archivos para lograr una mayor organización y reutilización del código).

En la función principal debe implementar un menú de opciones donde el usuario pueda elegir qué algoritmo utilizar.

Resumen

- Un algoritmo de ordenamiento reorganiza los elementos de una lista en orden creciente o decreciente. (burbuja, selección, inserción)
- Un algoritmo de búsqueda encuentra un elemento dentro de una estructura de datos. (búsqueda lineal)
- La elección de un algoritmo depende de su **eficiencia**, la cantidad de datos, del contexto en que se aplican, de su estabilidad (mantiene el orden relativo de los elementos iguales)
- Modifique los algoritmos con otros tipos de datos.



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 3:

Algoritmos de Ordenamiento y Búsqueda II

Contenido

1. Eficiencia de Algoritmos: tiempo - espacio
 - Notación O grande.
2. Algoritmo de Mezcla (Merge Sort)
3. Algoritmo Rápido (Quick Sort)
4. Algoritmo de Búsqueda Binaria (Binary Search)

1. Eficiencia de algoritmos

- Una forma fundamental de comparar algoritmos es analizar **cuánto esfuerzo computacional requieren** para resolver una tarea (**eficiencia**).
- Para describir este esfuerzo utilizamos la notación Big O (O grande) que nos permite expresar como crece el **tiempo de ejecución** (o el **uso de memoria**) en función del tamaño de la entrada, considerando los diferentes casos de desempeño:
 - ✓ **Peor caso:** El rendimiento del algoritmo bajo las condiciones más desfavorables. Big O describe, en general, el peor caso posible del comportamiento de un algoritmo.
 - ✓ **Caso promedio:** El rendimiento esperado en condiciones típicas o promedio de entrada.
 - ✓ **Mejor caso:** El rendimiento del algoritmo en las condiciones más favorables

Ejemplos comunes de la notación O grande

- **$O(1)$ orden constante:** indica que tendremos el mismo rendimiento sin importar el tamaño de entrada.
- **$O(\log(n))$ orden logarítmico:** indica que el tiempo de ejecución aumenta linealmente mientras que el tamaño n crece de forma exponencial.
- **$O(n)$ orden lineal:** indica que la complejidad del algoritmo aumenta de manera proporcional al tamaño del arreglo.
- **$O(n \log(n))$ orden lineal-logarítmico:** indica que el tiempo de ejecución crece proporcional a n veces el logaritmo de n.
- **$O(n^2)$ orden cuadrático:** el tiempo de ejecución crece proporcional al cuadrado del tamaño de entrada.
- **$O(2^n)$ orden exponencial:** el tiempo se duplica con cada incremento en el tamaño n.

Ejemplos:

Dado un arreglo de tamaño n, estimar la eficiencia de un algoritmo para:

- **Determinar si el primer elemento de un arreglo es igual al segundo**
 - ✓ Requiere solo una comparación. Es independiente de n. $O(1)$
- **Determinar si el primer elemento de un arreglo es igual a cualquiera de los otros elementos.** $O(n)$
 - ✓ Requiere n-1 comparaciones (n domina)
- **Determinar si algún elemento está duplicado en el arreglo** $O(n^2)$
 - ✓ Requiere $n(n-1)/2$ comparaciones (n^2 domina)

¿Qué mide O grande?

El crecimiento del tiempo de ejecución de un algoritmo en relación con la cantidad de elementos procesados.

Un algoritmo que requiere n^2 comparaciones:

- Si $n = 4$, el algoritmo requerirá 16 comparaciones $\cancel{4} \times 4$
- Si $n = 8$, 64 comparaciones.

Un algoritmo que requiere $\frac{n^2}{2}$ comparaciones

- Si $n = 4$, el algoritmo requerirá 8 comparaciones $\cancel{4} \times 4$
- Si $n = 8$, 32 comparaciones.

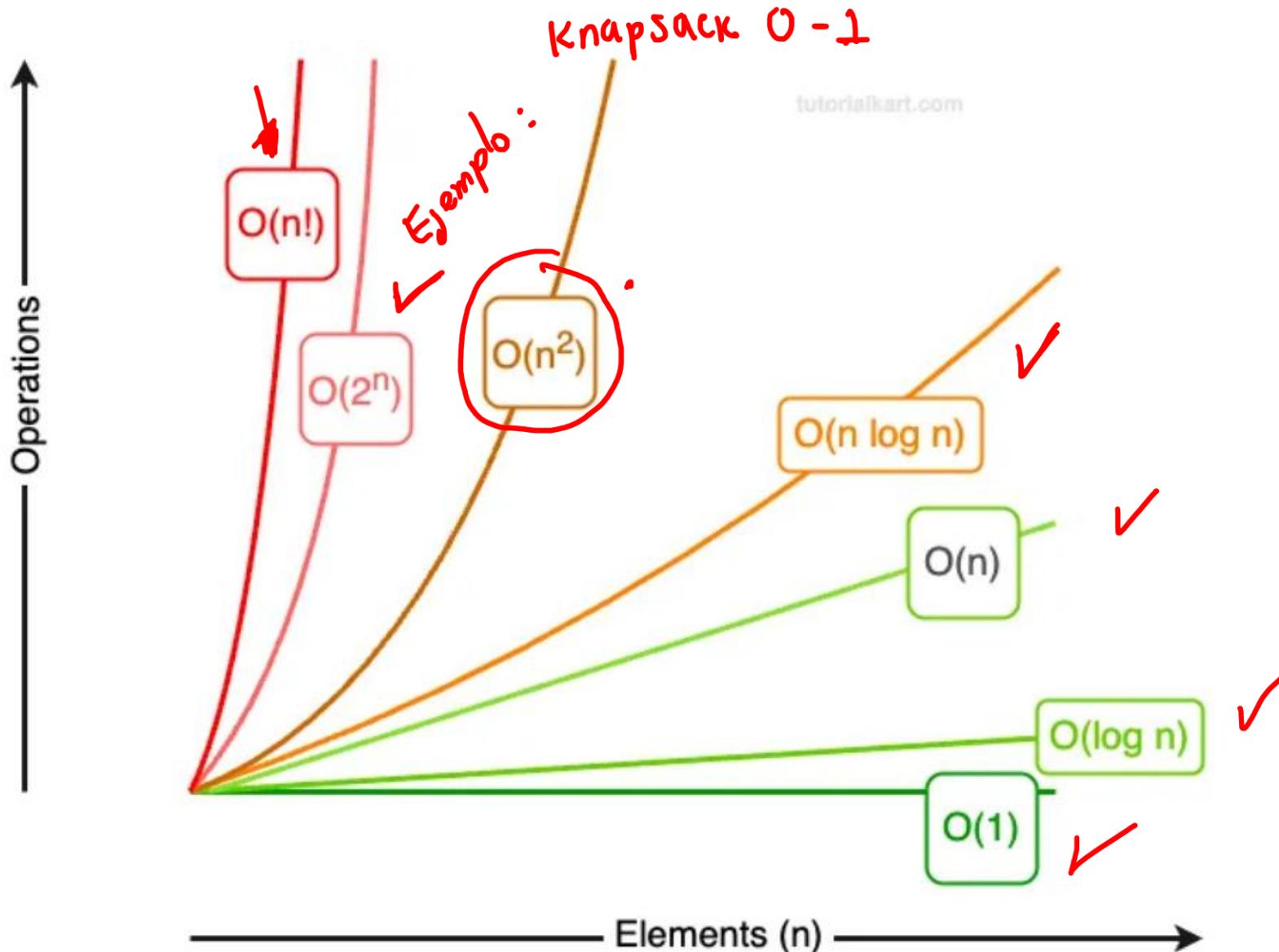
En ambos algoritmos al duplicar el número de elementos se cuadriplica el número de comparaciones.

Ambos algoritmos crecen como el cuadrado de n , por lo que O grande ignora la constante y ambos algoritmos se consideran $O(n^2)$.

Ejercicios

- Eficiencia del algoritmo de Búsqueda Lineal. $\underline{O(n)}$
- Eficiencia del algoritmo de Burbuja $O(n^2)$
- Eficiencia del algoritmo de Ordenamiento por selección $O(n^2)$
- Eficiencia del algoritmo de Ordenamiento por Inserción $O(n^2)$

Comparativa de los órdenes de complejidad



Método Divide y Vencerás

- Este enfoque consiste en descomponer un problema complejo en varios subproblemas más pequeños y manejables, que son versiones simplificadas del problema original.
- Estos subproblemas se resuelven recursivamente y, una vez resueltos, se combinan sus soluciones para obtener la solución completa del problema original.
 - ✓ **Caso base:** Se resuelve directamente sin necesidad de recurrencia.
 - ✓ **Caso recursivo:** Se lleva a cabo en tres etapas clave:
 - **1 División:** El problema se descompone en uno o más subproblemas de tamaño reducido.
 - **2 Conquista:** Se resuelven los subproblemas de forma recursiva.
 - **3 Combinación:** Se combinan las soluciones de los subproblemas para construir la solución al problema original.

Ejercicio: ③

Implementar una función que reciba dos arreglos ordenados y los combine en un solo arreglo ordenado. Ejemplo:

Input:

```
arr1 = {1, 3, 5, 7};  
arr2 = {2, 4, 6, 8};
```

Output:

```
arr = {1, 2, 3, 4, 5, 6, 7, 8}
```

2. Ordenamiento por mezcla (Merge sort)

- Es un algoritmo basado en el enfoque de Divide y Vencerás.
- Es un algoritmo estable (mantiene el orden relativo de los elementos con valores iguales).
- Tiene una complejidad temporal de $O(n \log n)$ lo que lo convierte en uno de los algoritmos más eficientes para ordenar grandes cantidades de datos.

¿Cómo Funciona?

- **Dividir:** El arreglo se divide recursivamente en dos mitades.
- **Ordenar:** Cada mitad se ordena de manera recursiva.
- **Mezclar:** Se fusionan las dos mitades ordenadas para obtener el arreglo completo ordenado.

Ejemplo: Merge Sort

Arreglo a ordenar (en forma creciente):

99	6	86	15	58	35	86	4	0
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

Dividir

99	6	86	15	58	35	86	4	0
-----------	----------	-----------	-----------	-----------	-----------	-----------	----------	----------

99	6	86	15
-----------	----------	-----------	-----------

58	35	86	4	0
-----------	-----------	-----------	----------	----------

99	6
-----------	----------

86	15
-----------	-----------

58	35
-----------	-----------

86	4	0
-----------	----------	----------

99	6
-----------	----------

86	15
-----------	-----------

58	35
-----------	-----------

86

4	0
----------	----------

4	0
----------	----------

-

-

-

-

-

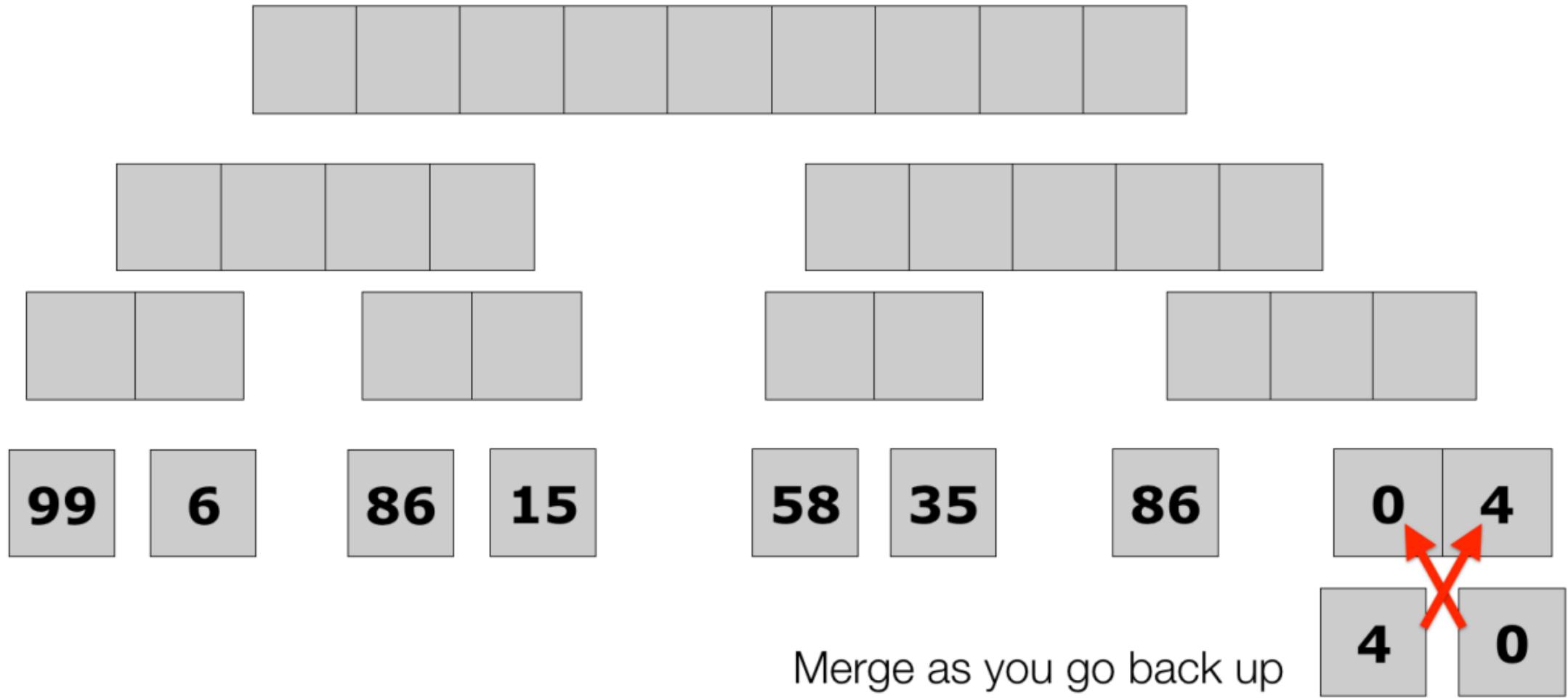
-

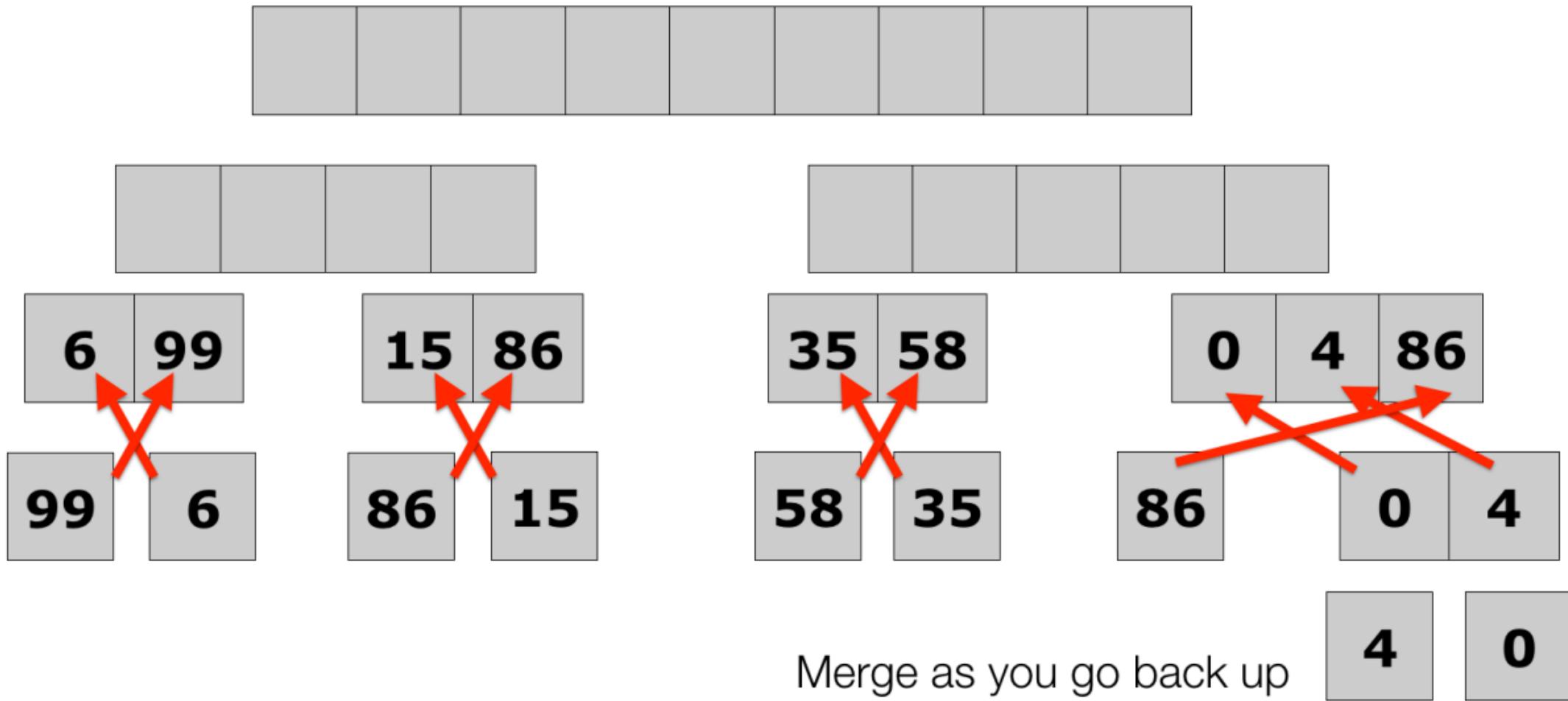
-

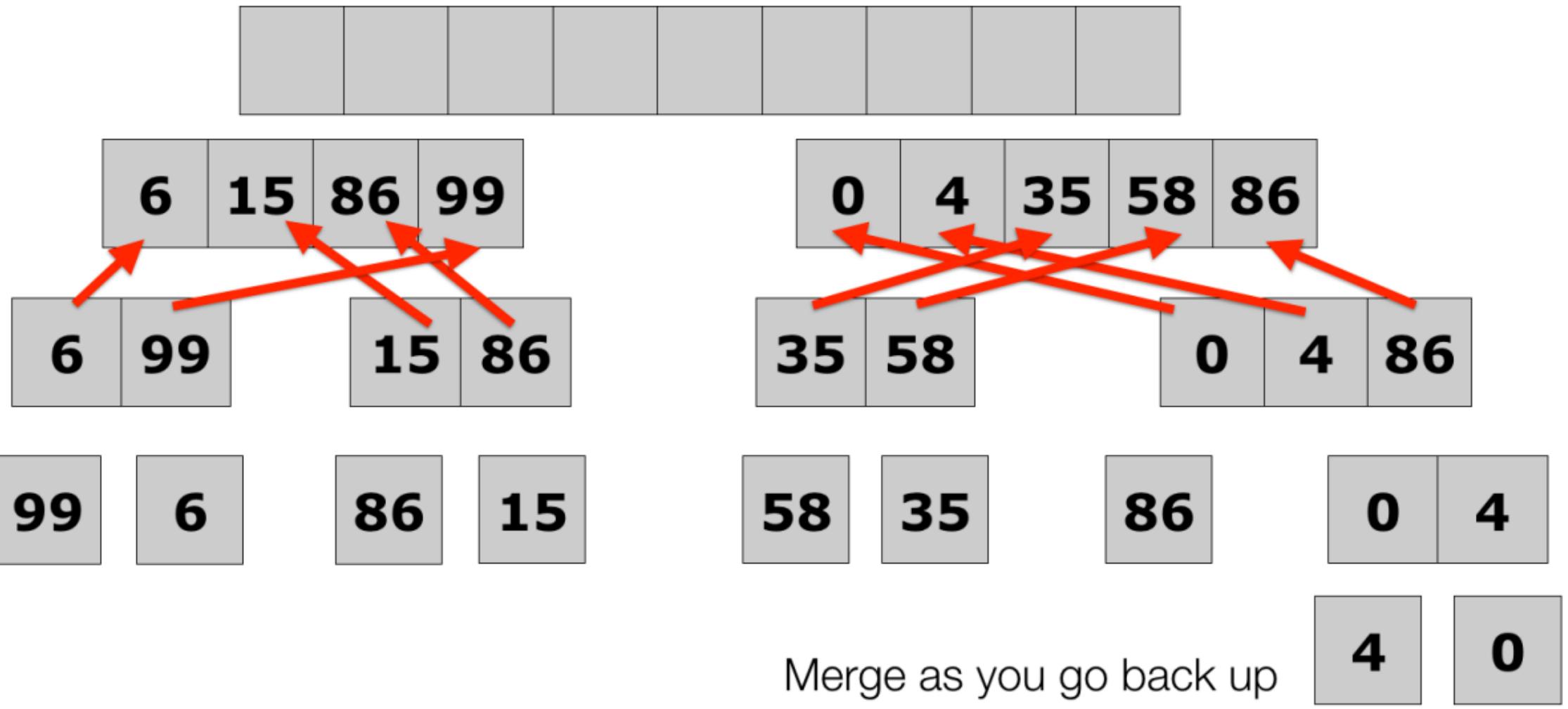
-

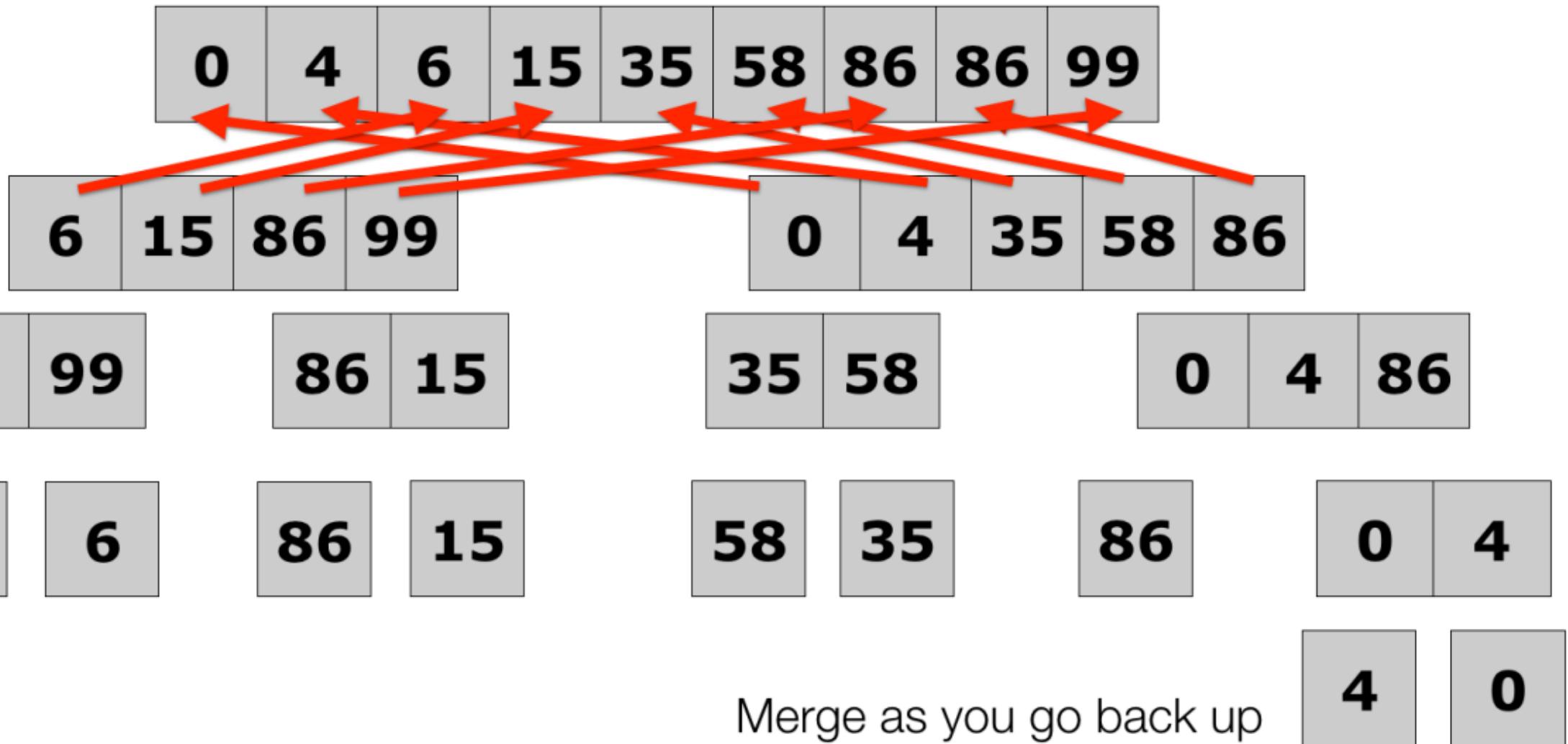
-

Ordenar y fusionar:









Merge Sort recursivo en C++

Ver repositorio de la semana 3

3. Ordenamiento rápido (Quick Sort)

- Es otro algoritmo de ordenamiento eficiente que utiliza la estrategia Divide y Vencerás. Creado por Tony Hoare en 1960.
- Es uno de los algoritmos más populares debido a su eficiencia en la práctica. Tiene una complejidad promedio de $O(n \log n)$, pero su rendimiento depende de la elección del pivote.

¿Cómo Funciona?

- **Elegir un pivote**: el elemento pivote puede ser el primer, último, medio, o un pivote aleatorio.
- **Particionar**: Se reorganiza el arreglo colocando los elementos menores que el pivote a su izquierda y los mayores a su derecha.
- **Recursión**: El algoritmo se aplica recursivamente a las dos mitades del arreglo (a la izquierda y derecha del pivote).

Observaciones

- La elección del pivote tiene un gran impacto en la eficiencia del algoritmo. Elegir un pivote de manera aleatoria ayuda a evitar peores casos ($O(n^2)$).
- Es un algoritmo in-place (no necesita memoria adicional significativa como Merge Sort)
- En sistemas con poca memoria la recursión profunda puede causar un desbordamiento de pila.
- A diferencia de Merge Sort, Quick Sort no es un algoritmo estable (elementos con valores iguales pueden cambiar su orden relativo),

Ejemplo: Quick Sort

- Arreglo a ser ordenado:

6	5	9	12	3	4
----------	----------	----------	-----------	----------	----------

pivot (6)



Elegir pivote y particionar

6	5	9	12	3	4
---	---	---	----	---	---

pivot (5)

5	3	4
---	---	---

pivot (9)

6

9	12
---	----

< 5

3	4
---	---

> 5

5

X

< 9

6

<

--

> 9

9

12

 pivot (3)

3	4
---	---

5

--

6

--

9

12

< 3

--

3

4

5

--

6

--

9

12

> 3

3	4	5	6	9	12
---	---	---	---	---	----

Quick Sort recursivo en C++

Ver repositorio de la semana 3

Ejercicio: adivinar un número

- Implemente un programa en C++ que permita al usuario adivinar un número aleatorio generado por la computadora en un rango de 1 a 100.
- Durante la ejecución, el programa brinda retroalimentación al usuario, indicando si el número ingresado es **demasiado alto, demasiado bajo o correcto**. El ciclo de adivinanzas continúa hasta que el usuario acierta el número.

¿Cuántos intentos son necesarios para adivinar el número? Responda la misma pregunta cuando el rango es entre 1 y 1000

Para optimizar la cantidad de intentos, es fundamental comprender el funcionamiento del algoritmo de **Búsqueda Binaria**, ya que este permite reducir significativamente el número de pruebas necesarias.

5. Búsqueda Binaria

- Es un algoritmo eficiente para encontrar un elemento dentro de un arreglo ordenado.  *aplicar búsqueda binaria a un arreglo circular.*
- El principio básico es dividir el arreglo en mitades de forma iterativa y reducir el espacio de búsqueda a la mitad en cada paso.
- La búsqueda binaria tiene una complejidad temporal $O(\log n)$, lo que la hace mucho más eficiente que la búsqueda lineal ($O(n)$). *wwww*

¿Cómo funciona? Calcular el índice medio = $(\text{inicio} + \text{fin})/2$;

- Comparar el valor buscado con el valor medio. Si coinciden, el algoritmo finaliza.
- Si el valor medio es mayor que el valor buscado, el nuevo rango de búsqueda será la mitad izquierda del arreglo.
- Si el valor medio es menor que el valor buscado, el nuevo rango de búsqueda será la mitad derecha del arreglo
- Repetir el proceso hasta que el valor sea encontrado o el rango de búsqueda se reduzca a cero ($\text{inicio} > \text{fin}$). ↳ *(inicio <= fin)*

¿Por qué es más eficiente?

- Cada vez, divide por 2 la cantidad de datos que quedan por buscar.
- **Recordemos el ejercicio “adivinar un número”:** Si empezamos con 100 números, tendremos que adivinar como máximo 7 veces !.
Ejemplo:

Número a adivinar 1 : 50 -> 25 -> 12 -> 6 -> 3 ->1

Número a adivinar 100: 50 -> 75 -> 88 -> 93 -> 96 ->98 -> 100

- Podemos calcular el número de intentos para adivinar el número?
Ejemplo con 16 elementos, si adivinamos incorrectamente hasta llegar a un elemento. $16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ $16 * \left(\frac{1}{2}\right)^4 = 1$

¿Por qué es más eficiente?

- En general: para n elementos $n * \underbrace{\left(\frac{1}{2}\right)^k} = 1$, donde k es el número de veces que debemos dividir.

De donde: $k = \log_2(n)$

- Si $n = 1000$, $k = \log_2(1000) = 9.96 \approx 10$ (máximo número de pasos.)

Ejemplo: Búsqueda Binaria

Search 23	0	1	2	3	4	5	6	7	8	9
	2	5	8	12	16	23	38	56	72	91

23 > 16
take 2nd half

L=0	1	2	3	M=4	5	6	7	8	H=9	
	2	5	8	12	16	23	38	56	72	91

23 < 56
take 1st half

0	1	2	3	4	L=5	6	M=7	8	H=9	
	2	5	8	12	16	23	38	56	72	91

Found 23,
Return 5

0	1	2	3	4	L=5, M=5	H=6	7	8	9	
	2	5	8	12	16	23	38	56	72	91

Una comparación entre la búsqueda secuencial y binaria puede observarse [aquí](#).

Búsqueda Binaria en C++

Ver repositorio de la semana 3

¿Versión recursiva?

Resumen

- **Complejidad temporal**: tiempo que tarda un algoritmo en función del tamaño de la entrada.
- **Complejidad espacial**: mide la cantidad de memoria usada por el algoritmo.
- **La notación Big-O** se utiliza para expresar la peor complejidad del algoritmo.
- **Merge Sort**: Divide el arreglo en dos mitades. Ordena ambas mitades de manera recursiva. Fusiona las mitades ordenadas.
 - ✓ Complejidad: Tiempo: $O(n \log n)$ (siempre).
Espacio: $O(\underline{n})$ (espacio para la fusión).
 - ✓ Estabilidad: Conserva el orden relativo de los elementos iguales.

- **Quick Sort**: Selecciona un pivote. Reorganiza los elementos para que los menores que el pivote queden a la izquierda y los mayores a la derecha. Recursivamente ordena los subarreglos.

- ✓ Complejidad: Temporal: $O(n \log n)$ (mejor caso, caso promedio)
 $O(n^2)$ (peor caso, pivote desequilibrado).
Espacio: $O(\log n)$ para la recursión.
- ✓ Inestabilidad: Puede cambiar el orden relativo de elementos iguales.

- **Búsqueda Binaria**: Algoritmo eficiente para arreglos ordenados. Divide el arreglo en dos mitades y compara el elemento buscado con el valor en el punto medio. Repite el proceso con la mitad correspondiente.

- ✓ Complejidad: Tiempo: $O(\log n)$ (ideal para arreglos grandes).
- ✓ Espacio: $O(1)$ (en su versión iterativa).



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 4:

Punteros I

Contenido

1. Introducción a la Memoria y Variables
2. ¿Qué es un Puntero?
3. Declaración de Punteros
4. Operadores de Referencia (&) y Desreferencia (*)
5. Puntero a Puntero (Doble Puntero)
6. Punteros y Cadenas de Caracteres
7. Punteros Nulos y buenas prácticas
8. Puntero y Arreglos unidimensionales
9. Aritmética de Punteros.
10. Eficiencia en el uso de Punteros
11. Referencias vs Punteros

1. Introducción a la Memoria y Variables

Cada variable declarada en C++ tiene :

- ✓ **Nombre**, identificador para referirnos a una variable
- ✓ **Tipo**, obligatorio en C++. int, float, char, etc.
- ✓ **Dirección de memoria**, ubicación física en la memoria RAM donde se almacena el valor de la variable.

¿Cómo accede la computadora a una variable?

Cada vez que usamos el nombre de una variable en nuestro código, la computadora realiza **internamente** los siguientes pasos:

1. **Busca la dirección de memoria** asociada al nombre de la variable.
2. **Accede a esa dirección** para recuperar o modificar el valor almacenado.

¿Podemos hacer esto nosotros?

¡Sí! En C++, los punteros nos permiten acceder directamente a la dirección de memoria de una variable y manipular su contenido.

C++ nos permite realizar cualquiera de estos pasos de forma independiente con los operadores & y *:

1. **&x** obtiene la dirección de memoria de la variable x.
2. ***&x** accede a esa dirección de memoria y **recupera el valor almacenado en ella**, es decir, desreferencia la dirección de x.

Ejercicio: Declara una variable float, muestre su valor y dirección de memoria

2. Punteros

- Un **puntero** es una variable que **almacena la dirección de memoria** donde se encuentra almacenado otra variable.
Es decir, los punteros "apuntan" a la ubicación en la memoria de otra variable.
- Podemos imaginar un puntero como una llave que abre un casillero.
La llave (el puntero) no tiene el contenido del casillero, sino la ubicación de ese casillero. Al usar la llave, podemos acceder o modificar el contenido guardado en él.



3. Declaración de un puntero

La declaración de un puntero sigue la siguiente sintaxis, en este orden:

- Tipo de dato del puntero (int, float, char, etc.).
- El operador asterisco *.
- Un nombre para el puntero.

Ejemplo: `int *ptr; //no inicializado`

Buenas Prácticas:

- ✓ Inicializa puntero (al menos a nullptr).
- ✓ Usar nombres descriptivos.

Ejercicio: Declara un puntero a char y haz que apunte a una variable char.
Imprima su dirección.

4. Operadores de Referencia (&) y Desreferencia (*)

- & obtiene la **dirección de memoria** de una variable

Ejemplo `int x = 3;`
`int *p = &x;`

- * accede al valor apuntado

Ejemplo `cout << *p; // imprime el valor apuntado por p;`

- Además de acceder al valor, **también podemos modificarlo** a través del puntero desreferenciado.

Ejemplo `*p = 5; // modifica el valor de x a 5`
`// usando el puntero`

Ejercicio: Crea una función que reciba un puntero y modifique el valor original de la variable.

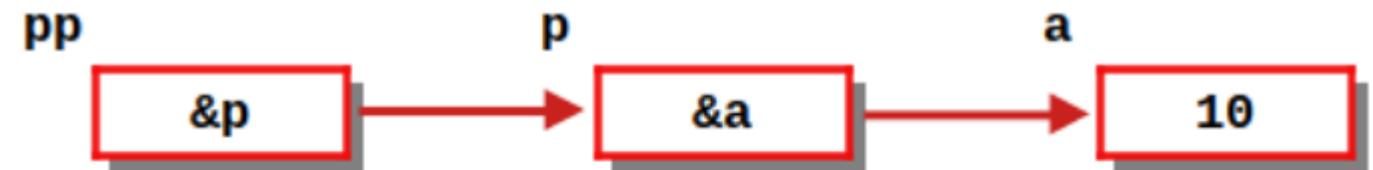
Ejercicio: Implemente una función en C++ que reciba como parámetro un **puntero a entero** y calcule el **cuadrado del valor al que apunta**.

5. Puntero a Puntero (Doble puntero)

- En C++, es posible declarar un puntero que **apunte a otro puntero**. A esto se le conoce como un **puntero a puntero**, y se declara usando **dos asteriscos (**)**.

```
int a = 5;  
int *p = &a;  
int **pp = &p;
```

```
cout << *p << endl;  
cout << **pp << endl;  
cout << *p << endl;
```



puntero que
apunta al puntero
que apunta al
dato

puntero que
apunta al dato

dato

Ejercicio: Modifica el valor de a desde pp

6. Punteros y Cadenas de caracteres

Cuando trabajamos con cadenas de caracteres en C++, es común utilizar una **matriz de caracteres (arreglo bidimensional)** para almacenar varias cadenas.

```
char nombres[3][6] = { "Alan",
                      "Bob",
                      "Carol"};  
  
for (int i = 0; i < 3 ; i++){
    cout << nombres[i] << endl;
}
```

0,0 char 'A'	0,1 char 'l'	0,2 char 'a'	0,3 char 'n'	0,4 char '\0'	0,5 char '\0'
1,0 char 'B'	1,1 char 'o'	1,2 char 'b'	1,3 char '\0'	1,4 char '\0'	1,5 char '\0'
2,0 char 'C'	2,1 char 'a'	2,2 char 'r'	2,3 char 'o'	2,4 char 'l'	2,5 char '\0'

¿Qué sucede si no conocemos la longitud que los nombres tendrán? ...

- Si en lugar de Carol tenemos ChristopherJones? Podemos asignar espacio adecuado (por ejemplo 20). Pero, esto provoca que se desperdicie más memoria:

0,0 char 'A'	0,1 char 'I'	0,2 char 'a'	0,3 char 'n'	0,4 char '\0'	0,5 char '\0'	0,6 char '\0'	0,7 char '\0'	0,8 char '\0'	0,9 char '\0'	0,10 char '\0'	0,11 char '\0'	0,12 char '\0'	0,13 char '\0'	0,14 char '\0'	0,15 char '\0'	0,16 char '\0'	0,17 char '\0'	0,18 char '\0'	0,19 char '\0'
1,0 char 'B'	1,1 char 'o'	1,2 char 'b'	1,3 char '\0'	1,4 char '\0'	1,5 char '\0'	1,6 char '\0'	1,7 char '\0'	1,8 char '\0'	1,9 char '\0'	1,10 char '\0'	1,11 char '\0'	1,12 char '\0'	1,13 char '\0'	1,14 char '\0'	1,15 char '\0'	1,16 char '\0'	1,17 char '\0'	1,18 char '\0'	1,19 char '\0'
2,0 char 'C'	2,1 char 'h'	2,2 char 'r'	2,3 char 'i'	2,4 char 's'	2,5 char 't'	2,6 char 'o'	2,7 char 'p'	2,8 char 'h'	2,9 char 'e'	2,10 char 'r'	2,11 char 'J'	2,12 char 'o'	2,13 char 'n'	2,14 char 'e'	2,15 char 's'	2,16 char '\0'	2,17 char '\0'	2,18 char '\0'	2,19 char '\0'

Aquí es donde los punteros permiten **gestionar memoria con mayor precisión** y trabajar con cadenas de forma más **segura y eficiente**.

Cuando utilizamos **punteros para manejar cadenas de caracteres**, no es necesario especificar el tamaño de las columnas

- Cada elemento del arreglo es un **puntero que apunta al primer carácter de una cadena**.

```
const char* nombres[] = { "Alan", "Bob", "ChristopherJones" };
```

En C++, cadenas literales (como "Alan"), estas se almacenan en una zona de memoria de solo lectura: **const char*** protege esas cadenas y le dice al compilador que no se deben modificar.

Ejercicio: Declare un **puntero a char** e **inicialícelo con una cadena literal**. Luego, **muestre el valor** al que apunta el puntero.

- Note que no tuvimos que incluir ningún valor de índice (evitamos el desperdicio de memoria). Basta reservar suficiente memoria para la creación de 3 punteros.



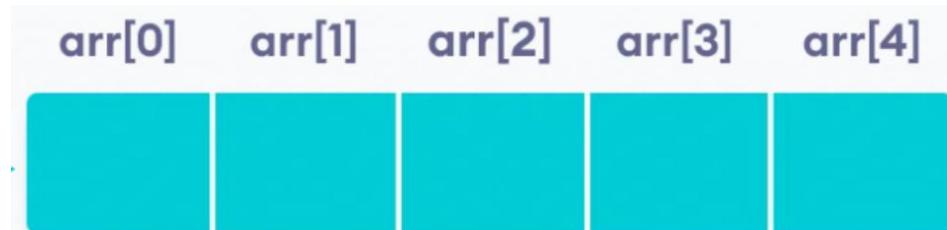
7. Punteros nulos y buenas prácticas

- Al igual que cualquier otra variable, los punteros deben **inicializarse correctamente**, ya sea al momento de declararlos o mediante una asignación posterior
- Un puntero **nulo** es aquel que **no apunta a ninguna dirección de memoria válida**. En C++, esto se representa tradicionalmente con 0 o NULL, pero desde C++11 se recomienda utilizar **nullptr**, ya que es más seguro y específico para punteros.
- Intentar acceder al valor apuntado por un puntero nulo (desreferenciarlo) provoca **comportamiento indefinido**.

8. Punteros y Arreglos unidimensionales

- El **nombre de un arreglo** representa un **puntero constante al primer elemento** del arreglo.

ptr = arr



- arr[0] se traduce **internamente** como: *(arr + 0), es decir, **el elemento en la posición inicial del arreglo**. Por ello, en C++, los arreglos están indexados desde cero.
- Cuando se pasa un arreglo a una función, en realidad **se pasa un puntero al primer elemento**, no una copia del arreglo completo.

Ejercicio: Escribe una función que reciba un arreglo como puntero y devuelva la suma de sus elementos.

Ejemplo: relación entre punteros y arreglos

```
int vector[5] = {1, 2, 3, 4, 5};

int *pv = vector; //pv es un puntero al primer elemento

cout << vector << endl; //devuelve la dirección del primer elemento

cout << &vector[0] << endl; // devuelve la dirección del primer elemento
```

Ejercicio: Implementar una función para inicializar y mostrar un arreglo de reales utilizando punteros. Mostrar además las direcciones de cada uno de sus elementos

9. Aritmética de Punteros

- Permite moverse entre ubicaciones en la memoria, generalmente entre elementos de un arreglo. Tenemos las siguientes **operaciones permitidas:**
- **Suma de un puntero y un entero**, agregar un número entero n a un puntero produce un nuevo puntero que avanza n posiciones en la memoria.

```
int arr [] = {6, 0, 9, 6, 5, 4};
```

```
int * ptr = arr;
```

```
ptr++; // nos movemos al próximo elemento del arreglo (no al próximo byte)  
//en el compilador: arr + 1 x sizeof(int)
```

```
int *ptr2 = arr + 3; //mueve el ptr2 para apuntar al cuarto elemento  
//en el compilador: arr + 3 x sizeof(int)
```

- **Resta de un puntero y un entero**, restar un número entero n a un puntero, retrocede n posiciones en la memoria
- **Resta de dos punteros**, devuelve la cantidad de elementos del tipo dado que caben entre los dos punteros.
 $\text{ptr2} - \text{ptr1}$
- **Comparación de punteros**, se pueden comparar mediante operadores relacionales como "`==`", "`<=`", "`>=`" y "`!=`"

Ejercicio: Usando aritmética de punteros implemente una función para invertir un arreglo de enteros.

Intercambiabilidad entre Arreglos y Punteros

- En muchas situaciones se puede usar el nombre del arreglo como si fuera un puntero. Recíprocamente si `int* ptr = arr;` podemos usar `p` como si se tratase de un arreglo.
- `ptr[3]` puede expresarse como `*(ptr + 3)`

Ejercicio: Implemente una función que reciba un arreglo y trabaje directamente con el mismo.

10. Eficiencia de los Punteros

- Ayudan a evitar copias innecesarias, permiten la manipulación directa de datos, se usa con frecuencia en estructuras de datos complejas
- Acceder a los elementos de un arreglo utilizando punteros en lugar de los índices usuales es más eficiente.

```
double data [10] = {0}, a;  
data[3] = 3.14;  
a = data[3];
```

```
double data [10] = {0}, a;  
*(data + 3) = 3.14;  
a = *(data + 3);
```

- Para calcular la posición de memoria de data[3] el compilador:
 1. Lee el valor de la dirección en la variable data.
 2. Calcula el incremento correcto (3×8 bytes)
 3. Agrega el incremento a data

En ambos casos tenemos el mismo número de operaciones ☺

```

double data [100];
int i;
for (i = 0; i < 100; i++) {
    if (data[i] < 0.) {
        cout <<"valor neg:\n " <<data[i];
    }
}

```

```

double data [100], *pd , * lastpd;
lastpd = data + 100;
for (pd = data; pd < lastpd; pd++) {
    if (*pd < 0.) {
        cout << "Valor neg:\n" << *pd;
    }
}

```

- Requiere calcular la dirección de data[i] para cada if:
 - 100 sumas: data + i
 - 100 multiplicaciones: i x 8 bytes.
- No necesitamos calcular en ninguno de los if para acceder al elemento actual *pd del arreglo
- La variable auxiliar lastpd sirve para evitar calcular 100 veces la suma data + 100 en la condición para salir del ciclo for.

Utilizando punteros obtenemos menor número de operaciones 😊

11 Referencias vs punteros

- En C y C++, se utiliza & para **obtener la dirección de memoria** de una variable. Sin embargo, en C++, el mismo símbolo & adquiere un **significado adicional cuando se utiliza en una declaración**: sirve para declarar una **referencia**, es decir, un **alias** para una variable existente.

- Ejemplos:

```
int y;
```

```
int &x = y; //x es una referencia (alias) de y
```

```
int jose = 5;
```

```
int &pepe = jose; //pepe es un alias de jose
```

```
void f(int &x){...}           f(y)
```

Aunque ambos permiten acceder indirectamente a otras variables, tienen **diferencias importantes en su uso y comportamiento**:

1. Las referencias están implícitamente desreferenciadas

- ✓ No es necesario usar el operador * para acceder al valor de una referencia.
- ✓ Se comportan como un alias directo de la variable original.

2. Las referencias no se pueden reasignar

- ✓ Una vez inicializada, una referencia no puede cambiar para referenciar otra variable.
- ✓ Un puntero sí puede apuntar a diferentes ubicaciones a lo largo del programa.

3. Las referencias deben inicializarse al declararse

- ✓ No puedes declarar una referencia sin asignarla a una variable existente.
- ✓ Los punteros pueden declararse sin inicialización (aunque no es recomendable).

Ejemplos: Punteros vs referencias C / C++

```
//Los punteros pueden ser  
//inicializados posteriormente  
int *ptr_i; // Declaración  
ptr_i = &i; // Inicialización
```

```
//Los punteros pueden cambiar de valor  
//en cualquier momento  
int *ptr_n = &x; // "ptr_n" apunta a la  
variable "x"  
ptr_n = &y;  
// Ahora "ptr_n" apunta a la variable "y"
```

```
//Las referencias no pueden almacenar un puntero nulo  
int *ptr = nullptr;  
int &reference = nullptr // ¡ERROR!
```

```
//Las referencias deben inicializarse al  
//momento en que se crean  
int i = 4;  
int &ref_i = i; // "ref_i" es el alias de la  
// variable "i"  
int &ref_j; // ¡ERROR! A "ref_j" se le debe  
// asignar alguna variable
```

```
//Las referencias no pueden  
//reasignarse a otra variable  
int x = 1;  
int y = 2;  
int &ref_n = x;  
// "ref_n" es un alias de "x"  
ref_n = y;  
// ¡ERROR! "ref_n" está unido de por vida a "x"
```

Ejercicios

- Dado el arreglo `int numeros[5]{11, 22, 33, 44, 55};` imprimir sus elementos utilizando punteros
- Para `int numeros[8]{11, 22, 33, 44, 55, 66, 77, 88};` defina 2 punteros a algún elemento del arreglo y calcule su diferencia (número de elementos entre dos punteros)
- Para `int numeros[8]{11, 22, 33, 44, 55, 66, 77, 88};` defina 2 punteros a algún elemento del arreglo y compárelos utilizando los operadores relacionales
- Implemente dos funciones: una que use referencia y otra puntero, ambas deben modificar el valor original.

Resumen:

- El operador * se utiliza de dos maneras diferentes:
 1. **Declaración un puntero**, * se coloca después del tipo y antes del nombre de la variable. `int *ptr;`
 2. **Desreferencia**, * se coloca antes del nombre del puntero para desreferenciarlo, para acceder o modificar el valor al que apunta.
`*ptr; *ptr = 4;`
- De forma similar el operador &, puede usarse como:
 1. **Referencia**, para indicar un tipo de dato por referencia
`int &x = y;`
 2. **Dirección**, para tomar la dirección de una variable
`int *ptr = &x;`

- Los punteros, mediante el uso de aritmética de punteros, permiten:
 - recorrer manualmente arreglos,
 - controlar directamente la memoria,
 - acceder de forma eficiente a estructuras de datos,
 - implementar estructuras dinámicas (listas, árboles, etc.),
 - manipular datos sin copiar grandes bloques de memoria y
 - optimizar el rendimiento en operaciones de bajo nivel.
- Sin embargo, su uso incorrecto puede provocar errores graves, como:
 - acceso fuera de los límites del arreglo,
 - punteros colgantes (dangling pointers),
 - desreferenciación de punteros nulos (nullptr),
 - modificaciones involuntarias de datos y
 - dificultad para depurar errores relacionados con memoria.



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 5:

Punteros II

Contenido

1. Paso de punteros como parámetros de una función.
 - ✓ Paso por valor, por punteros, por referencia
2. Arreglo de punteros
3. Punteros a punteros
4. Punteros y matrices
5. Prioridad de los operadores *, () y []
6. Puntero void, puntero constante
7. Punteros inteligentes (Smart pointers)

1. Paso de punteros como parámetros de una función.

- **Paso por valor:** Copia el valor. No modifica el original.
- **Paso por puntero:** Permite modificar el valor original.
- **Paso por referencia:** Similar al puntero, pero con sintaxis más clara

Ejemplos:

```
void fporValor(int n);
```

```
void fporPuntero(int* n);
```

```
void fporReferencia(int& n);
```

2. Arreglo de punteros

- Es una colección de direcciones de memoria (punteros); por ejemplo

```
int *a[5];
```

```
char *b[10];
```

```
const char *frutas[] = {"Manzana", "Fresa", "Cereza"};
```

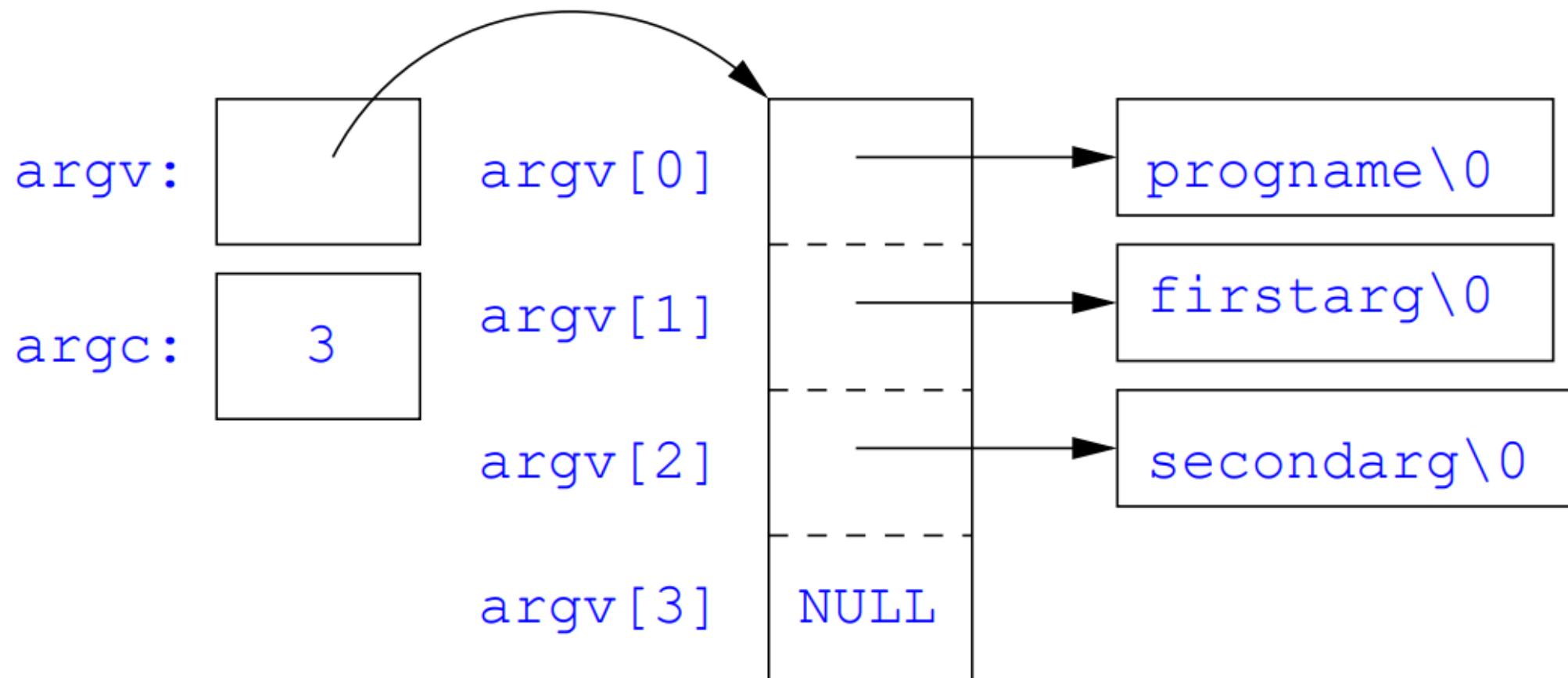
- Los arreglos de punteros son **útiles con cadenas**. Ejemplo: main con parámetros

```
int main(int argc, char *argv[])//o char **argv  
//....  
return 0;  
}
```

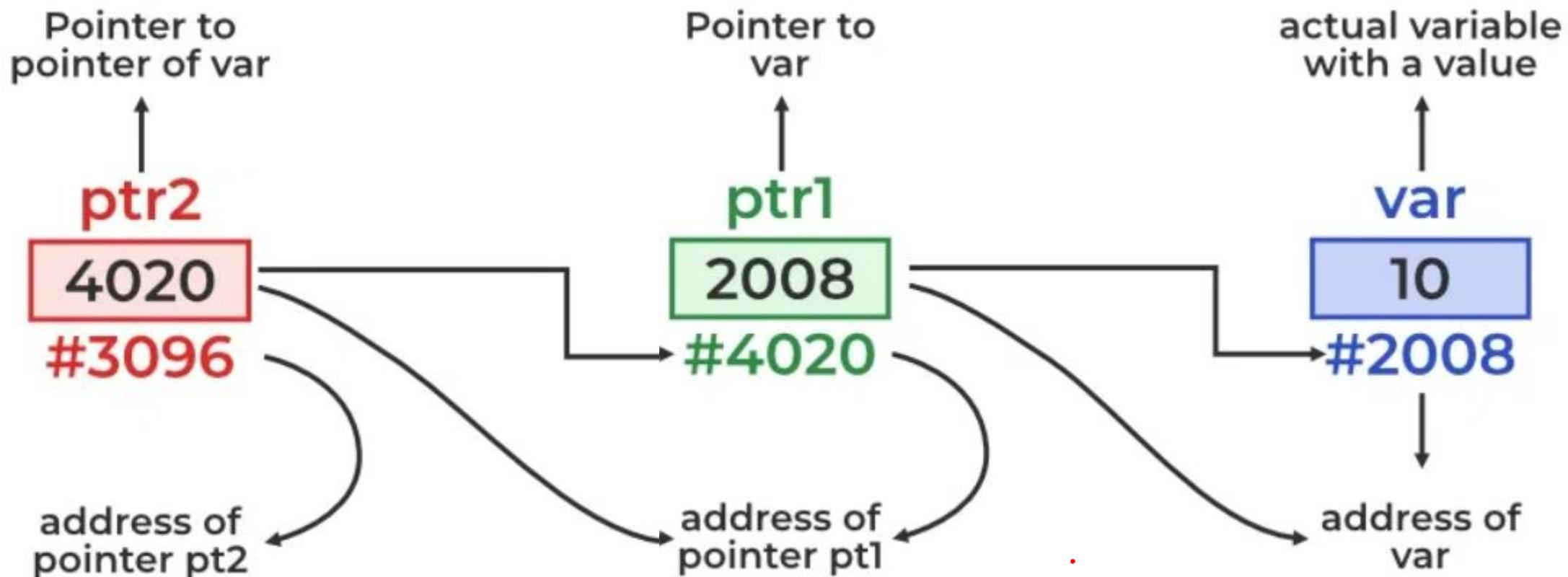
El primer argumento es el número de argumentos de línea de comandos y
El segundo es una lista de argumentos de línea de comandos

Ejemplo: arreglo de punteros

- argv es un arreglo de punteros a caracteres
- argc le indica al programador la longitud del arreglo



3. Punteros a punteros



Ejemplo: Punteros a punteros

```
int arr[3];
int *ptr,**pptr; // declaración de punteros
arr[0] = 10;
ptr = arr; //
pptr = &ptr;//

cout << &arr[0] << endl;//
cout << *&ptr << endl; //
cout << *pptr << endl; //      *pptr == *&ptr == ptr
cout << **pptr << endl; // **pptr==*( *pptr)==*(ptr)==arr[0]
cout << arr << endl; //
cout << &arr << endl; //
```

4. Punteros y matrices

```
int matrix[2][3] = {{1,2,3},{4,5,6}};
```

Un arreglo bidimensional (matriz) **se trata como un arreglo de arreglo**,

		Column
		0 1 2
Row	0	1 2 3
	1	4 5 6

matrix[0][0]	100	1
matrix[0][1]	104	2
matrix[0][2]	108	3
matrix[1][0]	112	4
matrix[1][1]	116	5
matrix[1][2]	120	6

Relación entre punteros y matrices

Dado la matriz: `int a[2][4] = {{1,2,3,4},{5,6,7,8}};`

- a **es la dirección del primer subarreglo** // `a == &a[0]` tipo: `int(*)[4]`.
- a[0] es la primera fila, // `a[0] == &a[0][0]` tipo `int[4]`
- a + 1 internamente se traduce en sumar el tamaño del subarreglo
`(4 * sizeof(int))` // `a + 1 == &a[1]` tipo: `int(*)[4]`
- *(a + 1) desreferencia el puntero a + 1, accediendo a a[1] (tipo `int[4]`)
// `*(a + 1) == a[1]`

- `*(a+1) + 1;` apunta al segundo elemento del subarreglo `a[1]`
`// *(a + 1) + 1 == &a[1][1]` es un puntero a int

- `*(*(a + 1) + 1);` desreferenciamos la dirección anterior
`// *(*(a + 1) + 1) == a[1][1] == 6`

Expresión	Equivalente	Significado
<code>a</code>	<code>&a[0]</code>	Dirección del primer subarreglo (<code>int (*)[4]</code>)
<code>a[0]</code>	<code>&a[0][0]</code>	Dirección del primer entero (<code>int*</code>)
<code>a + 1</code>	<code>&a[1]</code>	Dirección del segundo subarreglo
<code>*(a + 1)</code>	<code>a[1]</code>	Segundo subarreglo (<code>int[4]</code>)
<code>*(a + 1) + 2</code>	<code>&a[1][2]</code>	Dirección del 3er elemento de 2da fila
<code>*(*(a + 1) + 2)</code>	<code>a[1][2]</code>	Valor: 7

En general dada una matriz int $a[N1][N2]$, tenemos las siguientes formas equivalentes de desreferenciar uno de sus elementos $a[i][j]$:

1. $*(*(\mathbf{a} + \mathbf{i}) + \mathbf{j})$
2. $*(\mathbf{a}[\mathbf{i}] + \mathbf{j})$
3. $*(&\mathbf{a}[0][0] + N2 * i + j)$

Expresiones como: $\mathbf{a}[\mathbf{i}] == *(\mathbf{a} + \mathbf{i}) == *(\mathbf{i} + \mathbf{a}) == \mathbf{i}[\mathbf{a}]$
son equivalentes y tienen sentido en C++.

Ejemplo

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

int main() {
    int a[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8},{9, 10, 11, 12} };

    // Mostrar dirección
    cout << "Direccion de a          : " << a << endl;
    cout << "Direccion de a[0]        : " << a[0] << endl;
    cout << "Direccion de a + 1       : " << (a + 1) << endl;
    cout << "Direccion de a[1]        : " << a[1] << endl;

    // Restar dos punteros
    cout << "\n(a + 1) - a = " << (a+1) - a << " (filas de distancia)" << endl;

    return 0;
}
```

Pasando matrices a funciones

- Al pasar una matriz bidimensional a una función, la primera dimensión (filas) no es obligatoria, ya que **lo que realmente se pasa es un puntero al primer subarreglo (fila)**. Por eso, las siguientes formas son equivalentes y válidas:

```
void f(int a[5][10]) { ... } //Paso filas y columnas
```

```
void f(int a[][10]) { ... } // equivale a int(*a)[10]
```

```
void f(int (*a)[10]) { ... } //Es lo que sucede internamente
```

```
void f(int **a) { ... } //Válido sólo para matrices creadas como  
arreglo de punteros
```

Punteros a funciones

- El **nombre de una función representa la dirección de inicio del código** de esa función en memoria; esto significa que **podemos asignar el nombre de una función a un puntero a función** y luego **invocar esa función a través del puntero**.
- Un puntero a función puede almacenar la dirección de una función con una firma específica (mismo tipo de retorno y los mismos parámetros).
- **Declaración:** `int (*f) (int, int);`
`//f es un puntero a una función que recibe dos int y retorna un int.`
- Note que: `int *f(int, int);`
Es una declaración de **una función que retorna un puntero a int**

Ejercicio: escriba un programa que calcule el máximo de 2 números utilizando puntero a función y una función que retorne puntero

5. Prioridad de los operadores *, (), y []

Las reglas de precedencia para las declaraciones, de mayor a menor, son:

1. **Paréntesis** () que agrupan partes de una declaración. Se utilizan para forzar un orden específico y evitar ambigüedad
2. **Operadores de sufijo** (mayor precedencia que *)
 - ✓ [] Acceso a un elemento de un arreglo
 - ✓ () Llamada a función.
3. **Operador de prefijo**
 - ✓ * Operador de desreferenciación (puntero a)

Ejemplo

```
char* (*(*f[5])(char*))[];  
  
//f es un  
  
//arreglo de 5 punteros  
  
//a funciones  
  
//que aceptan un puntero a char  
  
//y retornan un puntero  
  
//a un arreglo de punteros  
  
//a char
```

6. Punteros void

- Denominado también **puntero genérico**: Declaración: `void *ptr;`
- Puede almacenar la dirección de cualquier tipo de dato sin tener que hacer una conversión explícita (cast) y podemos convertirlo a cualquier tipo de dato en caso sea necesario.
- No podemos operar (**desreferenciar**) con el objeto señalado por el puntero void, ya que se desconoce el tipo.
- Tampoco podemos hacer aritmética de punteros.

Ejemplo: Punteros void

```
void *ptrVoid;
int a = 78, arr[20];
float r = 283.91, *ptrFloat;
char nombre[30] = "Javier";
ptrVoid = &a;// recibe la dirección de a
ptrVoid = arr;// apunta al primer elemento de arr
ptrVoid = &arr[5];// recibe la dirección de arr[5]
ptrVoid = &r;// recibe la dirección de r

(void*)nombre;
//para lograr imprimir la dirección

*ptrFloat = 456.78;
ptrVoid = ptrFloat; //apunta a la misma dirección que ptrFloat

ptrVoid = nombre;//recibe la dirección de la cadena nombre
```

6. Tipos de Punteros con const

const se usa para indicar que algo no debe modificarse. Existen 3 combinaciones:

1. **Puntero a dato constante:** `const int *ptr;` o `int const *ptr`
Los datos señalados NO SE PUEDEN cambiar.

```
int i1 = 8, i2 = 9;  
const int * iptr = &i1; //int const *iptr  
/*iptr = 9; error: assignment of read-only location  
iptr = &i2; // ok
```

2. **Puntero constante a dato:** `int *const ptr;`
Los datos apuntados PUEDEN cambiarse; pero el puntero NO SE PUEDE cambiar para que apunte a otros datos.

```
int i1 = 8, i2 = 9;
int * const iptr = &i1; // puntero constante
//debe ser inicializado durante la declaración
*iptr = 9; // ok
// iptr = &i2; error: assignment of read-only variable
```

3. Puntero constante a dato constante: `const int *const ptr;` los datos señalados NO SE PUEDEN cambiar; y el puntero NO SE PUEDE cambiar para que apunte a otros datos:

```
int i1 = 8, i2 = 9;
const int * const iptr = &i1;
// *iptr = 9; // error: assignment of read-only variable
// iptr = &i2; // error: assignment of read-only variable
```

Ejemplo: El nombre de un arreglo es un puntero constante

- Cuando declaramos un arreglo como: `int A[n];` el nombre del arreglo se comporta como un puntero constante al primer elemento
`// A es equivalente a: int* const`

```
int A[5] = {1, 2, 3, 4, 5};  
int* ptr = A; // Válido  
ptr = ptr + 1; // Válido: podemos mover ptr  
A = ptr; // Error
```

- Los arreglos tienen tamaño fijo. Se les asigna memoria en tiempo de compilación (en el stack). Al salir del bloque donde fueron definidos, la memoria se libera automáticamente.

7. Punteros inteligentes

- Es un mecanismo de C++ para resolver el problema de la gestión de memoria (desde la versión C++ 11 en adelante).
- El problema es que para alojar memoria en el heap debemos utilizar la palabra new y eliminar explícitamente las variables con el operador delete.
- Los punteros inteligentes permiten eliminar de la memoria automáticamente cuando el nombre de la variable sale de su ámbito.
- La biblioteca <memory> ofrece distintos tipos, lo más comunes son:
 - ✓ Punteros únicos std::unique_ptr
 - ✓ Punteros compartidos std::shared_ptr
 - ✓ Punteros débiles std::weak_ptr

Más sobre Smart pointers

- <https://docs.hektorprofe.net/cpp/12-punteros-inteligentes/>
- https://en.wikipedia.org/wiki/Smart_pointer
- <https://www.geeksforgeeks.org/smарт-pointers-cpp/>
- <https://youtu.be/8gUz60-GtPA>

Ejercicio: primera parte

Escribir un programa que lea un texto, luego proceda a dividir el texto en palabras (secuencia de caracteres separados por un espacio en blanco) y almacene estas palabras en un arreglo. Realizado el almacenamiento proceda a ordenar las palabras en orden alfabético. Sugerencia:

- I. Obtenga los caracteres uno por uno y verifique si se ha llegado al final de una cadena.
- II. Cargue los caracteres en una matriz bidimensional cuyo primer índice (filas) cuenta las palabras y el segundo (columnas) los caracteres que componen una sola palabra.
- III. Aplique un algoritmo de ordenamiento adecuado (burbuja, selección, inserción, merge sort, quick sort), para ordenar las palabras.

Ejemplo:

Input: bienvenidos al curso cc112

Debe almacenarse en una matriz y mostrarse en la forma:

bienvenidos

al

curso

cc112

Output:

al

bienvenidos

cc112

curso

Segunda parte

Escribir una segunda versión del ejercicio que involucre el uso de todas las herramientas vistas en el curso:

funciones, algoritmos de ordenamiento para arreglo de y punteros.

Detalles técnicos

- En `int a[3]`; a técnicamente no es un puntero. a es el nombre del arreglo, pero en la mayoría de las expresiones decae (se convierte implícitamente) en un puntero al primer elemento del arreglo. Por ejemplo, en **asignaciones o al pasar el arreglo a una función**
- El nombre del arreglo es una dirección fija y constante (no puede ser cambiado para apuntar a otro lugar). Un puntero normal puede ser modificado para apuntar a cualquier otra dirección.
- Para que un puntero apunte a un arreglo completo de 3 elementos, debe declararse explícitamente como `int (*ptr)[3]`

Detalles técnicos

- Ejemplos donde el nombre de un arreglo no decae en un puntero a su primer elemento
 1. El operador sizeof(a) no decae en un puntero al primer elemento

```
int a[5] = {1, 2, 3, 4, 5};  
cout << sizeof(a) << endl; // Devuelve 20 si cada entero ocupa 4 bytes.  
cout << sizeof(int*) << endl; // tamaño de un puntero  
  
//&a NO decae, Puntero al arreglo completo ie dirección del arreglo completo  
//a se convierte en un puntero al primer elemento  
//&a[0] dirección del primer elemento
```

2. El operador & aplicado a un arreglo no decae a un puntero al primer elemento, sino que devuelve la dirección del arreglo como un todo.

```
int (*ptr1)[5] = &a; // Puntero al arreglo de 5 elementos  
int *ptr2 = a; // Puntero al primer elemento
```

Detalles técnicos

3. Si tienes una referencia a un arreglo, el nombre del arreglo no decae a un puntero, ya que estás refiriéndote al arreglo completo.

```
int arr[5];
int (&ref)[5] = arr; // ref es una referencia al arreglo completo, no un puntero
```

4. Cuando pasas un arreglo a una función como referencia, el nombre del arreglo no decae a un puntero al primer elemento, sino que la referencia se trata como un tipo completo de arreglo.

```
void funcion(int (&arr)[5]) {
    cout << sizeof(arr); // Devuelve el tamaño del arreglo completo (20 bytes)
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    funcion(a); // Pasa el arreglo por referencia, no como puntero.
    return 0;
}
```

Detalles técnicos

5. En arreglos multidimensionales, el nombre del arreglo no decae inmediatamente a un puntero al primer elemento del primer subarreglo; en su lugar, decae progresivamente a punteros a subarreglos

```
int c[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
int (*ptr)[4] = c; // ptr es un puntero a un arreglo de 4 enteros

cout << (*ptr)[1]; // Imprime 2, el segundo elemento del primer subarreglo
```

En este caso, c decae a un puntero a un arreglo de 4 enteros (`int (*)[4]`), no a un puntero a un puntero (`int**`).



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 6:

Cadena de caracteres I

Contenido

1. Representación en memoria: Caracteres ASCII
2. Aritmética limitada de Caracteres
3. Arreglo y cadena de caracteres
4. Puntero y cadena de caracteres
5. Funciones de entrada de datos
6. Funciones para el manejo de caracteres I

Resumen: Memoria, punteros y arreglos

- **Punteros y memoria**
 1. Si `int v=10;` es una variable, entonces `&v` es su **referencia** (dirección).
 2. `int *p;` declara a `p` como un **puntero a int**.
 3. Si hacemos `p = &v`, entonces `*p` es `v`. (desreferencia)
- **Punteros y arreglos**
 1. Si `int a[N];` entonces `a` es un puntero a su primer elemento
 2. Si hacemos `p = a`, entonces `p+i` vale `&a[i]` y `*(p+i)` es `a[i]`.
 3. Los punteros a elementos de un arreglo se pueden incrementar `p++`, decrementar `p--` y comparar para moverse en un arreglo.

```
char c[] = "Hola";
c[0] = 'S';
//c++; ERROR el nombre de un arreglo es un puntero CONSTANTE
cout << c << endl;
cout << "sizeof(c) = " << sizeof(c) << endl; //tamaño del arreglo en bytes
```

```
const char *d = "Mundo";
d++; //aritmetica de punteros
cout << d << endl;
cout << "sizeof(d) = " << sizeof(d) << endl; //tamaño del puntero

cout << *d << endl; //desreferencia

cout << (void *)&d[0] << endl; //Conversión estilo C dirección del carácter

//La conversión explícita al estilo C++ seria
// reinterpret_cast<void*>(&d[0])
```

1. Caracteres ASCII.

- ASCII (American Standard Code for Information Interchange) es un estándar que asigna un valor numérico a 128 caracteres (0-127), incluyendo letras, números, signos de puntuación y caracteres de control. (1967, primera versión, ASCII estándar)
- En 1981, **IBM desarrolló una extensión** del código ASCII (“página de código 437”), donde se incorporaron 128 caracteres nuevos, como símbolos gráficos, letras acentuadas, signos de puntuación entre otros (0 - 255).
- En la actualidad, la mayoría de los sistemas informáticos utilizan el código ASCII para representar caracteres, símbolos, signos y textos.

Caracteres ASCII de control

00	NULL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	HT	(tab horizontal)
10	LF	(nueva línea)
11	VT	(tab vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc.vínculo datos)
17	DC1	(control disp. 1)
18	DC2	(control disp. 2)
19	DC3	(control disp. 3)
20	DC4	(control disp. 4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sínc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin del medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_		

ASCII extendido (Página de código 437)

128	ç	160	á	192	ł	224	ó
129	ü	161	í	193	ł	225	ß
130	é	162	ó	194	ł	226	ö
131	â	163	ú	195	ł	227	ò
132	ä	164	ñ	196	—	228	õ
133	à	165	Ñ	197	+	229	ö
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	Ł	232	þ
137	ë	169	®	201	Ł	233	Ú
138	è	170	¬	202	Ł	234	Ù
139	ï	171	½	203	Ł	235	Ù
140	î	172	¼	204	Ł	236	ý
141	ì	173	ı	205	=	237	Ý
142	Ä	174	«	206	+	238	—
143	Ä	175	»	207	¤	239	‘
144	É	176	„	208	ð	240	≡
145	æ	177	„	209	đ	241	±
146	Æ	178	„	210	Ê	242	≡
147	ô	179	—	211	Ê	243	¾
148	ö	180	—	212	È	244	¶
149	ò	181	Á	213	—	245	§
150	û	182	Â	214	—	246	÷
151	ù	183	À	215	—	247	—
152	ÿ	184	©	216	—	248	°
153	Ö	185	—	217	—	249	..
154	Ü	186	—	218	Ł	250	·
155	ø	187	—	219	Ł	251	·
156	£	188	—	220	—	252	³
157	Ø	189	¢	221	—	253	²
158	×	190	¥	222	—	254	■
159	f	191	—	223	—	255	nbsp

2. Aritmética limitada de caracteres

- En C++, los caracteres se pueden manipular como enteros, gracias a su correspondencia con los códigos ASCII. Para realizar la aritmética de caracteres debemos tener en cuenta que el rango de caracteres está entre -128 y 127 o entre 0 y 255 (unsigned char).

```
char ch1 = 125, ch2 = 10;  
ch1 = ch1 + ch2; //135  
cout << (int)ch1 << endl; // >127  
cout << (char)(ch1 - ch2 - 4);
```

```
char c = 'd';  
if (c >= 'a' && c <= 'z') {  
    c = c - 32;  
}  
cout << c << endl; // Muestra 'D'
```

- **Ejercicio 1:** Escribir un programa que genere 20 caracteres de forma aleatoria y cuente el número de ocurrencias de cada carácter generado.

Cadenas



- Una cadena es un arreglo (colección) de caracteres (char), terminado en el carácter nulo '\0' (que vale 0) en la memoria.
- Una cadena es un **objeto** cuyo tipo string, está definido en el archivo <string>.
- En lo que sigue nos enfocaremos en cadenas al estilo C, debido a que aún se suelen utilizar en C++. Ejemplos de cadenas:
"Buenos Dias" //cadena con 2 palabras
"Juan" //Cadena de una palabra
"" //cadena vacía
Los ejemplos anteriores reciben el nombre de **cadena de literales**



Cadenas: Declaración e inicialización

- **Declaración:** Una cadena se declara como un arreglo de char que tenga espacio suficiente para todos los caracteres y el carácter nulo.

```
char s[10]; // almacena hasta 9 caracteres
```

```
char t[11]; // almacena hasta 10 caracteres
```

- **Inicialización:** al momento de su declaración.

```
char s [10] = "ejemplo"; // 7 + '\0' y sobran 2
```

```
char t[] = "ejemplo "; // arreglo de 8 char
```

En estas formas de inicialización el carácter nulo '\0' es **insertado automáticamente al final** de la cadena.

```
char s [10];// puede contener cualquier cosa
```

Cadenas: Declaración e inicialización

- También podemos inicializar de la siguiente forma:

```
char c[]={ 'e', 'j', 'e', 'm', 'p', 'l', 'o', ' ', '1', '\0' };
```

- ✓ Aquí la cadena es inicializada carácter a carácter.
- ✓ En este método el **carácter nulo debe ser ingresado por el programador**

Initialization	Memory representation
char animal[]="Lion";	L i o n '\0'
char location[]="New Delhi";	N e w D e l h i '\0'
char serial_no[]="A011";	A 0 1 1 '\0'

Ejemplo: intercambiar el contenido de 2 cadenas

1. Usando arreglo de caracteres

Si a y b son arreglos de caracteres y queremos intercambiar sus contenidos, debemos copiarlos de un lado a otro:

```
char a[] = "primero";
char b[] = "segundo";
char t[] ;

copiar (t, a); // copia "primero" a t
copiar (a, b); // copia "segundo" a a
copiar (b, t); // copia "primero" a b

//Implementar la función copiar
void copiar(char a[], char b[]) {
    //Complete aquí su código para copiar
    //cada carácter de b en a, no olvide
    //terminar con el carácter nulo
}

//Intercambiar el contenido de dos
cadenas llamando a la función copiar
```

```
void copiar(char a[], char b[]) {  
    int i = 0;  
    while(b[i] != '\0') {  
        a[i] = b[i];  
        i++;  
    }  
    a[i] = '\0';  
}
```

Ejemplo: intercambiar el contenido de 2 cadenas

2. Usando punteros a caracter

Si a y b son punteros a caracteres y queremos intercambiar las cadenas a las que apuntan, podemos realizarlo mucho más rápido:

```
char *a = " primero "
```

```
char *b = " segundo ";
```

```
char *t;
```

```
t = a; // t apunta a primero
```

```
a = b; // a apunta a segundo
```

```
b = t; // b apunta a primero
```

En este caso, sólo se intercambian los valores de los punteros.

//Ejercicio implemente su función intercambio
//para cadenas de caracteres usando punteros

```
void intercambio(const char **a, const char **b) {
    const char *t = *a; // t apunta a primero
    *a = *b; // a apunta a segundo
    *b = t; // b apunta a primero
}

int main(){
    const char *a = " primero ";
    const char *b = " segundo ";

    //llamar a la funcion intercambio
    intercambio(&a,&b);

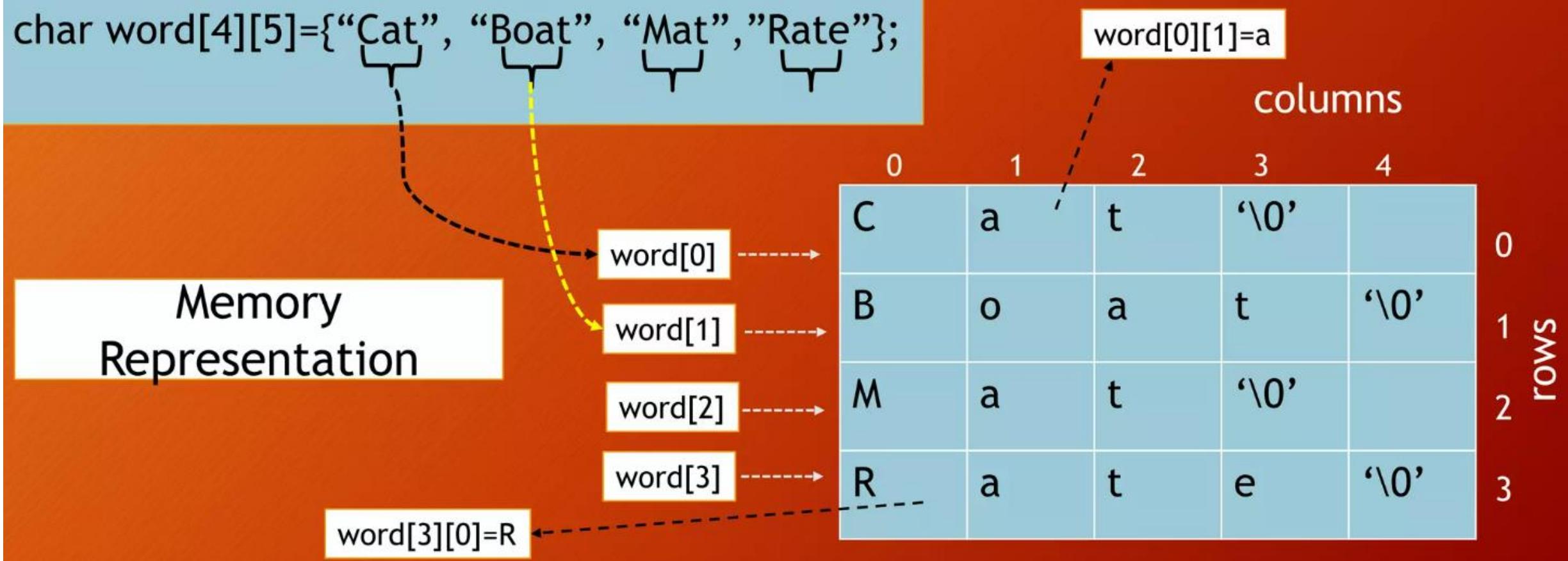
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

Arreglo de caracteres de 2 dimensiones

- De la misma forma que interpretamos un arreglo bidimensional, un arreglo de caracteres de dos dimensiones es un arreglo de cadenas (arreglo de arreglo de caracteres)
- Declaración: `char lista_alumnos[15][20];`
 - ✓ 15 es número de filas (alumnos) representa el número de cadenas.
 - ✓ 20 es el número de columnas, representa el tamaño de cada cadena (cada nombre puede contener hasta 19 caracteres)
- Inicialización: `char word[4][5] = {"Cat", "Boat", "Mat", "Rate"};`
- Acceso a sus elementos: `cout << word[2][0];` imprime M
`cout << word[0];` imprime Cat

Representación en memoria

```
char word[4][5]={“Cat”, “Boat”, “Mat”, ”Rate”};
```



Ejemplo: Arreglo de cadenas

- Declarar un arreglo de cadenas que contenga los nombres de los 7 días de la semana.

```
char dias[7][10] = {"lunes", "martes", "miercoles",
"jueves", "viernes", "sabado", "domingo"};
```

- Como arreglo adoptaría la siguiente forma.

```
dias[0] = "lunes"; dias[1] = "martes";
dias[2] = "miercoles"; ... ; dias[6] = "domingo";
```

- En la memoria se vería así:

```
"lunes0 .... martes0 ... miercoles0jueves0 ... viernes0 .. sabado0
...domingo0 .."
```

Arreglo de cadenas: Inconvenientes

- En un arreglo de este tipo, todas las cadenas ocupan la misma cantidad de caracteres. Esto quiere decir que cada una debe ser al menos tan larga como la cadena más larga. Por otro lado, las cadenas cortas desperdiciarán mucho espacio.
- Con frecuencia se desea mover una cadena de un lugar a otro en un arreglo (por ejemplo, para hacer copias o mantenerlas ordenadas).
- En este caso se deberá procesar cada carácter de la cadena y entre más largas será más lento.

4. Arreglo de Punteros a caracteres

- Una alternativa es declarar un arreglo de punteros a caracteres. Considerando el ejemplo anterior podría declararse así:

```
char *dias[7] = {"lunes", "martes", "miercoles",
"jueves", "viernes", "sabado", "domingo"};
```

- Esto se interpretaría exactamente igual que un arreglo:

```
dias[0] = "lunes"; dias[1] = "martes";
dias[2] = "miercoles"; ... ; dias[6] = "domingo";
```

- Pero en la memoria se vería distinto: dias[0] sería un puntero a la cadena "lunes" que ocupa sólo seis caracteres, etc.

5. Funciones de entrada de datos

- Una Para poder utilizar las funciones que controlan la entrada y salida estándar en C++ es necesario incluir la librería para flujos de entrada y salida de datos en C++: iostream y el espacio de nombres de la librería: std
- Salida estándar en C++:
La función cout junto con el operador de inserción << permite imprimir cadenas de texto y valores almacenados en variables.
- Entrada estándar en C++:
La función cin junto con el operador de extracción >>, permite capturar cualquier tipo de datos, **con excepción de cadenas de texto que contengan espacios**. Mientras que getline() permite la captura de textos con espacios.

Lectura (input) y escritura(output) de cadenas

- **Lectura:** podemos ingresar una cadena de 2 formas:
 1. Usando `cin >>`
 2. Usando la función `getline()`
- **Escritura:** Una cadena puede ser mostrada utilizando `cout <<`
- ¿Por qué no necesitamos un bucle para ingresar una cadena?
Porque **toda cadena tiene un delimitador**, el carácter nulo '\0'
- `cin >>` ingresa una cadena sin espacios, debido a que el operador de extracción `>>` detiene la entrada cuando encuentra un espacio en blanco

Lectura de cadenas: getline()

- La función getline() puede ser utilizada para ingresar texto de varias líneas
- Sintaxis: `cin.getline(cadena, MAX, Carácter delimitador)`, donde:
 - ✓ **cadena** es el arreglo de caracteres. Es la variable en la que se almacena la lectura.
 - ✓ **MAX** su valor determina el máximo número de caracteres permitidos durante la lectura.
 - ✓ **Carácter delimitador**: Es el carácter que, al ser encontrado en el proceso de lectura, detiene la entrada. El carácter delimitador por default es '\n'.
- La función getline continua con la lectura de la cadena hasta que el número máximo de caracteres permitidos es introducido o hasta encontrar el carácter delimitador.

6. Librería de funciones para cadenas

- Requiere el archivo de cabecera `<cstring>`
- Las funciones toman como argumentos una o más cadenas. Los argumentos pueden ser:
 - ✓ Nombre del arreglo de caracteres
 - ✓ Puntero a char
 - ✓ Literal de cadenas
- Podemos dividirlo en tres grupos:
 1. Para trabajar con cadenas: Requiere el archivo de cabecera `<cstring>`
 2. Para trabajar con caracteres: Requiere el archivo de cabecera `<cctype>`
 3. Para convertir datos numéricos y cadenas: requiere el archivo de cabecera `<cstdlib>`

6.1 funciones para cadenas

Agregar la librería de cabecera:

`#include <cstring>`

Para más detalles de otras funciones en la librería, visitar:

<https://cplusplus.com/reference/cstring/>

Función	Descripción
<code>strlen(s)</code>	Retorna la longitud (número de caracteres no nulos) de <code>s</code> .
<code>strcpy(s1, s2)</code>	Modifica <code>s1</code> reemplazando sus caracteres por copias de los caracteres de <code>s2</code> .
<code>strncpy(s1, s2, n)</code>	Modifica <code>s1</code> reemplazando sus caracteres por los <code>n</code> primeros caracteres de <code>s2</code> . Si <code>strlen(s2)</code> es menor que <code>n</code> , entonces <code>s1</code> es llenado con ‘ceros’ hasta que un total de <code>n</code> hayan sido escritos en él.
<code>strcat(s1, s2)</code>	Modifica <code>s1</code> concatenando <code>s2</code> al final de <code>s1</code> .
<code>strncat(s1, s2, n)</code>	Modifica <code>s1</code> concatenando con los <code>n</code> primeros caracteres de <code>s2</code> . Si <code>strlen(s2)</code> es menor que <code>n</code> , entonces <code>s1</code> es llenado con ‘ceros’ hasta que un total de <code>n</code> hayan sido concatenados al final de él.
<code>strcmp(s1, s2)</code>	Compara <code>s1</code> y <code>s2</code> , retornando un entero negativo, 0, o un entero positivo según <code>s1</code> es menor que, igual a, o mayor que <code>s2</code> .
<code>strstr(s1, s2)</code>	Busca en <code>s1</code> por la primera ocurrencia de <code>s2</code> , retornando un puntero al primer carácter de esta ocurrencia o <code>nullptr</code> si <code>s2</code> no es encontrado en <code>s1</code> .

6.2 funciones para el manejo de caracteres

Agregar la librería de cabecera: `#include <cctype>`

Función	Descripción
<code>islower(ch)</code>	Retorna <code>true</code> si <code>ch</code> es minúscula y <code>false</code> caso contrario.
<code>isupper(ch)</code>	Retorna <code>true</code> si <code>ch</code> es mayúscula y <code>false</code> caso contrario.
<code>tolower(ch)</code>	Retorna la minúscula equivalente a <code>ch</code> si <code>ch</code> es mayúscula.
<code>toupper(ch)</code>	Retorna la mayúscula equivalente a <code>ch</code> si <code>ch</code> es minúscula.

Existen otras funciones tales como: `isalpha(ch)` `isdigit(ch)`
`isalnum(ch)` `isspace(ch)` ...

Para mayor detalle de funciones en la librería visitar:

<https://cplusplus.com/reference/cctype/>

6.3 funciones para conversión de datos numéricos

Agregar la librería de cabecera: `#include <cstdlib>`

Función	Descripción
<code>atoi(s)</code>	Convierte <code>s</code> a un <code>int</code> (si es posible) y retorna este valor.
<code>atof(s)</code>	Convierte <code>s</code> a un <code>double</code> (si es posible) y retorna este valor.
<code>atol(s)</code>	Convierte <code>s</code> a un <code>long int</code> (si es posible) y retorna este valor.

Si `s` no contiene dígitos, **los resultados no están definidos:**

- La función puede devolver el resultado de la conversión hasta el primer dígito que no sea un dígito
- La función puede devolver 0

Para mayor detalle de funciones en la librería visitar:

<https://cplusplus.com/reference/cstdlib/>

Ejemplo

Escribir una función que permita convertir una cadena numérica literal a decimal. Utilice arreglo de caracteres, luego puntero a carácter.

```
//Con arreglos
int decimal ( char s[] ) {
    int n = 0;
    for ( int i = 0; '0' <= s[i] && s[i] <= '9'; i++)
        n = 10*n + (s[i] - '0');
    return;
}
```

Ejercicio: Escribir la versión con punteros y utilizando funciones de la librería de cadenas

Tokenización de cadenas (separar por delimitadores)

strtok divide una cadena en partes (tokens) utilizando delimitadores como espacios, comas, puntos, etc.

```
char* strtok(char* str, const char* delimitadores);
```

Ejemplo:

```
char texto[] = "uno,dos,tres,cuatro";
char* token = strtok(texto, ",");

while (token != nullptr) {
    cout << token << endl;
    token = strtok(nullptr, ",");
}
```

Más sobre cadenas

La clase **string** ofrece varias ventajas sobre las cadenas al estilo C:

- Gran cuerpo de funciones miembro

- Operadores sobrecargados para simplificar expresiones

Es necesario incluir el archivo de encabezado `#include <string>`

Para mayor detalle de la librería, visitar:

<https://cplusplus.com/reference/string/string/>



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 7:

Cadena de caracteres II

Contenido

1. Funciones para el manejo de cadenas
2. Funciones para caracteres
3. Funciones para convertir cadenas de caracteres

1. Funciones para el manejo de cadenas

Función	Descripción
strlen(s)	Retorna la longitud (número de caracteres no nulos) de s .
strcpy(s1, s2)	Modifica s1 reemplazando sus caracteres por copias de los caracteres de s2 .
strncpy(s1, s2, <u>n</u>)	Modifica s1 reemplazando sus caracteres por los n primeros caracteres de s2 . Si strlen(s2) es menor que n , entonces s1 es llenado con ‘ceros’ hasta que un total de n hayan sido escritos en él.
strcat(s1, s2)	Modifica s1 concatenando s2 al final de s1 .
strncat(s1, s2, n)	Modifica s1 concatenando con los n primeros caracteres de s2 . Si strlen(s2) es menor que n , entonces s1 es llenado con ‘ceros’ hasta que un total de n hayan sido concatenados al final de él.
strcmp(s1, s2)	Compara s1 y s2 , retornando un entero negativo, <u>0</u> , o un entero positivo según s1 es menor que, igual a, o mayor que s2 .
strstr(s1, s2)	Busca en s1 por la primera ocurrencia de s2 , <u>retornando un puntero al primer</u> carácter de esta ocurrencia o nullptr si s2 no es encontrado en s1 .

Debemos incluir la librería:

#include <cstring>

Ejemplo 1

Implemente una función que invierta una cadena (no utilice una cadena auxiliar)

```
void invertirCadena(char *cadena) {  
    char *inicio = cadena;  
    char *fin = cadena + strlen(cadena) - 1;  
    while (inicio < fin) {  
        char temp = *inicio;  
        *inicio = *fin;  
        *fin = temp;  
        //actualizamos los punteros  
        inicio++;  
        fin--;  
    }  
}
```

Ejemplo 2

Dado una lista de nombres almacenados en un arreglo de cadenas, utilice punteros a cadenas de caracteres y

1. Escriba una función que determine si un nombre está o no en el arreglo
2. Escriba una función que permita concatenar 2 nombres elegidos
3. Escriba una función que permita encontrar una subcadena

```
bool nombreEnArrreglo(const char *nombres[], int n, const char *nombre) {
    for (int i = 0; i < n; ++i) {
        if(strcmp(*(nombres + i), nombre) == 0){
            return true;
        }
    }
    return false;
}
```

Ejemplo 3

Dada la siguiente cadena: “--Aqui tenemos, un ejemplo de cadena.”, escribir una función que elimine los caracteres - , y .

La salida debe ser: “Aqui tenemos un ejemplo de cadena”

```
void eliminaCaracteres(char * cadena) {  
    char * recCadena = cadena;  
    char * resultado = cadena;  
    while (*recCadena != '\0') {  
        if (*recCadena != '-' && *recCadena != '.' && *recCadena != ',') {  
            *resultado = *recCadena;  
            resultado++;  
        }  
        recCadena++;  
    }  
    *resultado = '\0';  
}
```

2. Funciones para el manejo de caracteres

```
#include <cstring>
```

Función	Descripción
islower(ch)	Retorna true si ch es minúscula y false caso contrario.
isupper(ch)	Retorna true si ch es mayúscula y false caso contrario.
tolower(ch)	Retorna la minúscula equivalente a ch si ch es mayúscula.
toupper(ch)	Retorna la mayúscula equivalente a ch si ch es minúscula.

Ejemplo 4

Escribir una función que convierta una cadena de caracteres de modo que las minúsculas se convierten en mayúsculas y viceversa, mientras que los no alfabéticos se mantienen sin cambios.

Ejemplo

Entrada: "Veamos Su funCionamiento #@ 123! "

Salida: "vEAMOS sU FUNCIONAMIENTO #@ 123! "

3. Funciones para convertir cadenas

ASCII → integrar

Función	Descripción
atoi(s)	Convierte s a un int (si es posible) y retorna este valor.
atof(s)	Convierte s a un double (si es posible) y retorna este valor.
atol(s)	Convierte s a un long int (si es posible) y retorna este valor.

Ejemplo 5

Escribir un programa que reciba la siguiente cadena

"123, 45.67, -89, 1001, 23.45, 6789, 12.45"

y calcule la suma de todos los números enteros contenidos en ella

```
int sumarEnteros(char * cadena) {
    int suma = 0;
    char* token = strtok(cadena, ", ");
    while (token != nullptr) {
        double n = atof(token);
        if (floor(n) == n) {
            suma += n;
            cout << n << endl;
        }
        token = strtok(nullptr, ", ");
    }
    return suma;
}
```


Ejercicios

1. Escribe una función que [recorra](#) una cadena de caracteres utilizando un puntero y vaya imprimiendo cada carácter en la pantalla.
2. Implementa una función que calcule la [longitud](#) de una cadena de caracteres utilizando punteros.
3. Crea una función que [compare](#) dos cadenas de caracteres utilizando punteros y devuelva un valor que indique si son iguales o no.
4. Escribe una función que copie una cadena de caracteres en otra utilizando punteros.

Ejercicios

5. Implementa una función que concatene dos cadenas de caracteres utilizando punteros.
6. Escribe una función que invierta una cadena de caracteres utilizando punteros.
7. Implementa una función que elimine los caracteres duplicados de una cadena de caracteres utilizando punteros.
8. Escribe una función que cuente el número de palabras en una cadena de caracteres utilizando punteros.

Ejercicios

9. Implementa una función que reemplace todas las ocurrencias de una subcadena con otra subcadena en una cadena dada, utilizando punteros.
10. Escribe una función que invierta una cadena de caracteres utilizando punteros.
11. Implementa una función que elimine los caracteres duplicados de una cadena de caracteres utilizando punteros.
12. Escribe una función que cuente el número de palabras en una cadena de caracteres utilizando punteros.

Ejercicios

13. Implementa una función que reemplace todas las ocurrencias de una subcadena con otra subcadena en una cadena dada, utilizando punteros.
14. Escribe una función que tome una cadena de entrada y un delimitador, y devuelva un arreglo de punteros a tokens.
15. Modifica la función anterior para que cuente el número de palabras en lugar de simplemente imprimir los tokens.
16. Crea una función que busque un token específico en una cadena y lo reemplace por otro token dado.
17. Escribe una función que tome una cadena de entrada con palabras separadas por un delimitador y las ordene alfabéticamente.

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

void intercambio(char *a, char *b){
    char *temp;
    temp = a;
    a = b;
    b = temp;
}

void ordBurbuja(char *w[], int n){
    for(int i = 0; i < n - 1; i++){
        for(int j = 0; j < n - i - 1; j++){
            if(strcmp(w[j], w[j+1]) > 0){
                intercambio(w[j], w[j+1]);
            }
        }
    }
}

int main(){

    char v[][10] = {"Raul", "Luis", "Alberto", "Juan"};
    cout << sizeof(v) << endl;
    const char *w[] = {"Raul", "Luis", "Alberto", "Juan"};
    cout << sizeof(w) << endl;
    //ordBurbuja();

    return 0;
}
```



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 9:

Memoria dinámica con arreglos

Resumen del Curso

PRIMERA PARTE:

1. Recursividad e iteración
2. Ordenamiento y Búsqueda
3. Punteros
4. Cadenas de Caracteres

Resumen del Curso

SEGUNDA PARTE:

1. **Gestión Dinámica de Memoria:** Proceso de asignación y liberación de bloques de memoria durante el **tiempo de ejecución de un programa**
2. Estructura de Datos
3. Archivos
4. Introducción a la POO.

Contenido

1. Arreglos estáticos y Arreglos dinámicos
2. Memoria dinámica de la PC
3. Asignación de espacio: operador new
4. Liberación de espacio: operador delete
5. Arreglo de una dimensión
Uso de punteros y asignación dinámica de memoria
6. Arreglos multidimensionales
Uso de punteros y asignación dinámica de memoria

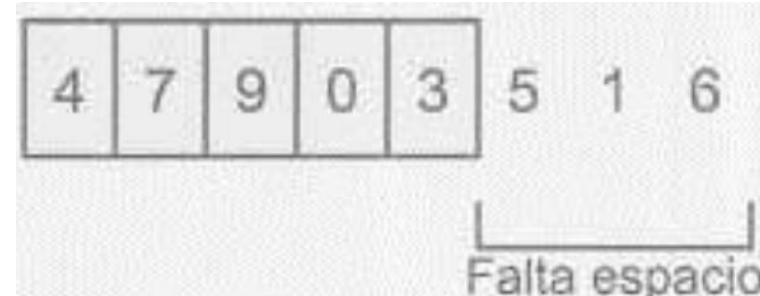
1. Arreglos Estáticos

```
const int CAPACIDAD = 10;  
double a[CAPACIDAD];
```

Tiene un **tamaño fijo**, definido en tiempo de compilación y no puede modificarse durante la ejecución del programa. El nombre de la variable, a, almacena la dirección del primer elemento

Inconvenientes:

- 1) Desperdicia memoria, cuando el tamaño del arreglo excede el número de valores que se almacenarán en ella.
- 2) Riesgo de desbordamiento, si se necesita almacenar más elementos de los que permite el tamaño fijo del arreglo , se corre el riesgo de sobrescribir memoria.



Necesidad de la memoria dinámica

La memoria dinámica es útil en situaciones donde:

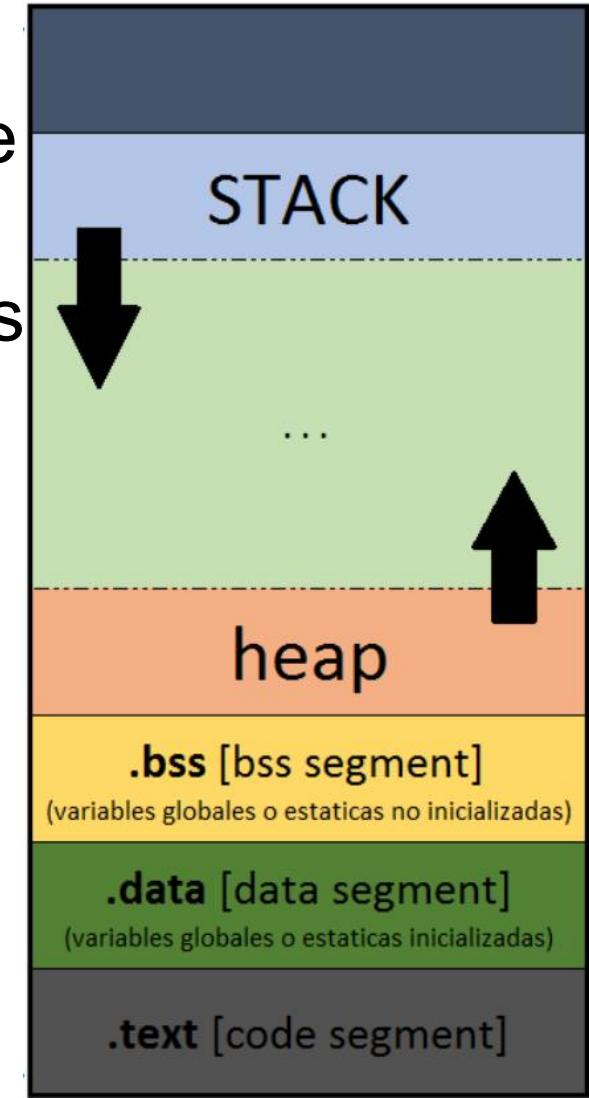
- El tamaño de los datos a almacenar no se conoce o varía en tiempo de ejecución
- Se desea optimizar el uso de memoria, asignando solo lo necesario
- Es necesario ampliar o reducir el tamaño del almacenamiento a medida que cambian las condiciones del programa.

El uso de **memoria dinámica** en C++ se gestiona de forma explícita:

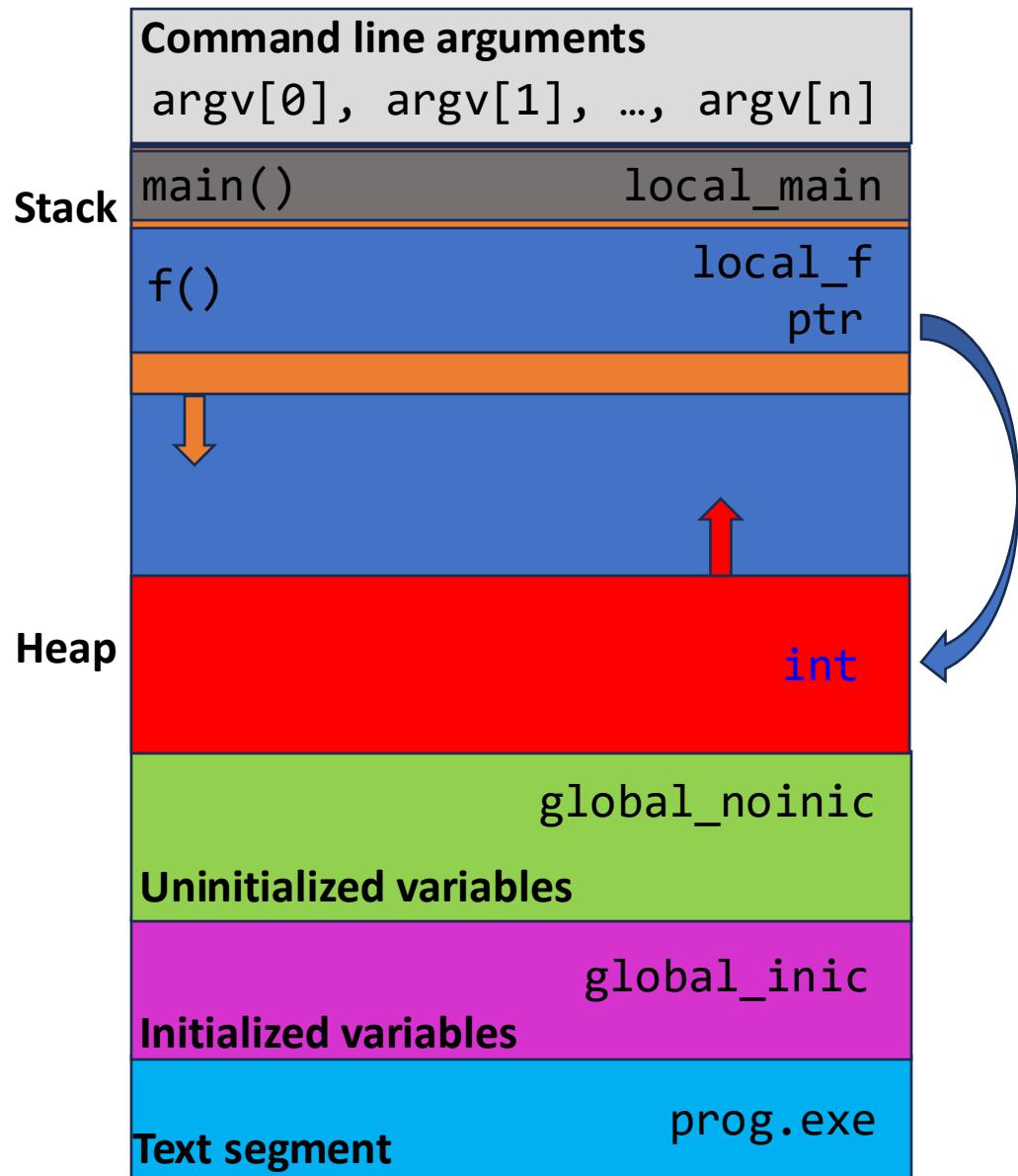
- La **asignación y liberación** se realiza mediante los operadores **new** y **delete** respectivamente.
- C++ moderno ofrece alternativas más seguras y recomendadas para la gestión de memoria.
 - Los Punteros inteligentes (Smart Pointers), que automatizan la liberación de memoria.
 - Los contenedores dinámicos como std::string, std::vector, etc que manejan internamente la memoria dinámica.

2. Memoria dinámica de la PC

- La porción de memoria que un Sistema Operativo asigna en la RAM a un proceso se divide, por lo general, en cuatro áreas distintas:
 1. **Área del programa** (text segment), contiene las instrucciones de máquina del programa; suele ser de solo lectura para evitar modificaciones accidentales.
 2. **Área de datos** (data segment), almacena constantes y variables globales o estáticas.
 3. **Pila** (Stack) utilizada para manejar llamadas a funciones (registro de activación). Almacena parámetros de funciones, variables locales, direcciones de retorno.
 4. **Montón** (Heap) destinada a la asignación dinámica de memoria.



Distribución de la memoria en un programa



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

int global_inic = 20; //segmento de datos
int global_noinic; //segmento bss (block started
                   by symbol)

void f(){
    int local_f; //stack (pila)
    int *ptr = new int; //heap
    *ptr = 20;
    delete ptr; //heap
}

int main(int argc, char *argv[]){
    int local_main; //Stack

    f();
    return 0;
}
```

Operadores de gestión de memoria en C++

- C++ introdujo nuevas formas de gestionar la memoria dinámica (respecto a la [gestión de memoria dinámica en C](#))
- Se agregaron dos nuevos operadores, llamados **new** y **delete**:
- **Ventajas** de new y delete:
 - ✓ new y delete forman parte del lenguaje C++, no son funciones de una librería
 - ✓ No es necesario especificar el tamaño de la memoria, este es determinado implícitamente por el tipo de datos: `int *p = new int;`
 - ✓ La asignación de memoria y la inicialización se pueden realizar en un solo paso: `double *q = new double { 3.1 }; // (3.1)`

3. Asignación de espacio en C++

Operador new:

- **Asigna espacios de memoria** en función del tipo de dato.
- Los datos pueden ser:
integrados (primitivos): int, float, double, char, bool, etc.
definidos por el usuario: struct, class, enum.
También, arreglos, matrices, objetos de clases STL
- El operador new **devuelve la dirección** de la ubicación de memoria asignada

Sintaxis:

```
tipo *ptr1 = new tipo
```

```
tipo *ptr2 = new tipo[tamaño] // Para arreglos
```

Ejemplo

```
int *pt;  
pt = new int; //asigna a pt la dirección de un bloque de  
//memoria de tamaño int  
  
char *cadena;  
int tam;  
cout << "Ingrese el tamaño del arreglo";  
cin >> tam;  
cadena = new char[tam]; //asigna a cadena la dir de un  
//arreglo de tipo char de tamaño tam
```

4. Liberación de espacio en C++

Operador delete: Libera el espacio de memoria asignado al puntero. Solo se puede emplear en aquellos punteros inicializados con new. **Note que el puntero sigue existiendo.**

Sintaxis:

```
int *pt1, *pt2;  
pt1 = new int; // Asigna a pt1 la dirección de un  
//bloque de memoria de tamaño int  
*pt1 = 130;  
pt2 = new int[5]; //asigna a pt2 la dir de un arreglo de  
// tipo int de tamaño 5  
delete pt1; //libera el bloque asignado a pt1  
delete [] pt2; //libera el arreglo asignado a pt2  
pt1=nullptr; pt2 = nullptr; //BUENA PRÁCTICA
```

Ejemplo

- Utilizando asignación dinámica y funciones. Escribir un programa que almacene las notas de n estudiantes del curso y calcule el promedio. A continuación, muestre el número de estudiantes cuya nota está por encima del promedio. Ejemplo de prototipo de las funciones

```
float* reservaMemoria(int );
```

```
void leerNotas(float*, int);
```

```
float promedio(float*, int);
```

```
void liberaMemoria(float*);
```

Fugas de memoria (memory leak)

- Ocurre cuando se asigna memoria usando el operador new y no se libera la memoria.
- El operador delete se utiliza para liberar un solo espacio de memoria
- El operador delete [] se usa para liberar un bloque (arreglo) de memoria
- Por cada new se debe usar un delete para liberar la memoria asignada
- Solo debemos reasignar memoria si se ha liberado antes
`char * s = new char [20];`
`s = new char[50]; //No! primero delete[] s;`

Fugas de memoria

- Cada variable dinámica debe asociarse con un puntero. Cuando la variable dinámica se desvincula de su puntero, **se pierde su dirección, por lo que es imposible liberar la memoria**

```
char* s1 = new char [20];
```

```
char* s2 = new char[40];
```

```
strcpy(s1, "Memory leak");
```

```
strcpy(s2, s1); //Si s2 = s1; 40 bytes no se pueden liberar
```

```
delete [] s2;
```

```
delete[] s1; //posible error de violación de acceso en  
//caso que s2 = s1;
```

```
s1 = nullptr; s2 = nullptr;
```

Verificación de la asignación de memoria

En ocasiones la memoria del Heap puede agotarse, por lo que no todos los requerimientos de asignación dinámica se pueden efectuar.

Hay dos formas de verificar:

1. La compilación lanza una excepción de tipo `bad_alloc`
2. Utilizar `nothrow`, si no es posible la asignación a través de `new`, devuelve `nullptr`.

```
int *p = new (nothrow) int [5];
```

5. Asignación en arreglos de una dimensión

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

int main(){
    int N;
    cout << "Ingrese el tamaño del arreglo: ";
    cin >> N;
    int* A = new int[N]; // asignar dinámicamente memoria de tamaño N
    for (int i = 0; i < N; i++){
        A[i] = i + 1; // asignar valores a la memoria asignada
    }
    for (int i = 0; i < N; i++) {
        cout << A[i] << " "; // o *(A + i) imprime el arreglo de 1D
    }
    delete[] A; // desasignar memoria
    A = nullptr;
    return 0;
}
```

6. Asignación en arreglos multidimensionales

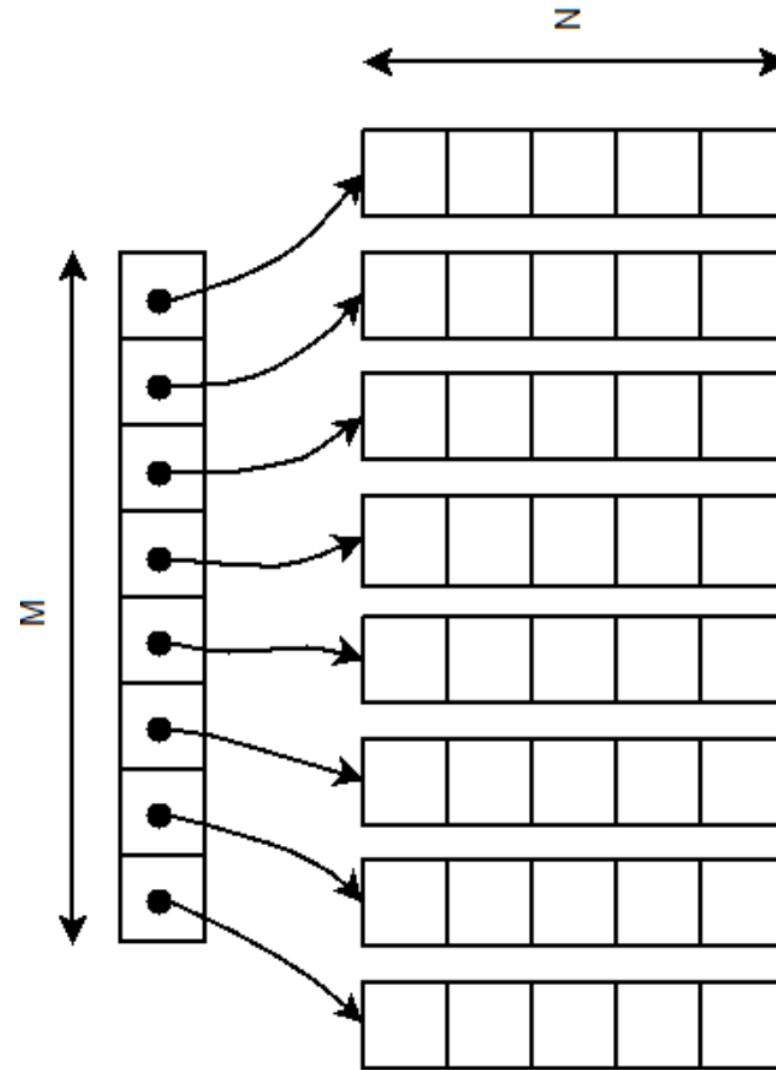
```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int M, N;
    cout << "Ingrese el número de filas (M) y columnas(N): ";
    cin >> M >> N;

    int* A = new int[M * N]; // Asignar memoria de tamaño M x N
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            *(A + i * N + j) = rand() % 10; // ó (A + i*N)[j]
        }
    }
    delete[] A; // Desasignar memoria
    A = nullptr;
    return 0;
}
```

Usando un arreglo de punteros

```
int *a[M];
```



Usando un arreglo de punteros

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

int main() {
    const int M = 4; // puedo usar constantes
    const int N = 5;

    int** A = new int*[M]; // Crear un arreglo de punteros de tamaño M

    // Asignar dinámicamente memoria de tamaño N para cada fila
    for (int i = 0; i < M; i++) {
        A[i] = new int[N];
    }

    // Desasignar memoria utilizando el operador delete
    for (int i = 0; i < M; i++) {
        delete[] A[i]; // Liberar la memoria de cada fila
    }
    delete[] A; // Liberar el arreglo de punteros
    return 0;
}
```

Ejercicio: Asignación para arreglos 3D

- Forma 1: Utilizando punteros
- Forma 2: Utilizando puntero a puntero a puntero

Números aleatorios racionales

```
#include <iostream>
#include <cstdlib> // rand() srand() RAND_MAX
#include <ctime> //time()
using namespace std;

int main(){
    cout << RAND_MAX<<endl;

    srand(time(0));
    for(int i = 0; i < 10; i++){
        int z = rand() % 10 + 1;
        cout << z <<" " ;
    }
    //Aleatorios racionales
    cout << endl;
    for(int i = 0; i < 10; i++){
        float z = (double)rand() / RAND_MAX * 10;
        cout << z <<" " ;
    }

    return 0;
}
```

Smart Pointers y asignación dinámica

- Los punteros inteligentes se pueden usar para asignar memoria dinámica en el heap.
- Gestionan automáticamente la vida útil de la memoria asignada de forma dinámica (es un ejemplo de [RAII](#))
- Permite evitar fugas de memoria ([memory leak](#)) y punteros colgantes ([dangling pointers](#)) que pueden ocurrir cuando la memoria no se gestiona adecuadamente.

Sobre operadores Bit a Bit

I. Investigar:

1. ¿Qué son los operadores bit a bit y cómo se diferencian de los operadores aritméticos?

2. Enumera y describe los operadores bit a bit disponibles en C++. Proporciona ejemplos de cada uno.

3. ¿Cuáles son las aplicaciones comunes de los operadores bit a bit en el desarrollo de software?

4. ¿Cómo afecta el uso de operadores bit a bit en el rendimiento de un programa?

Operadores Bit a Bit

II. Ejercicios de programación

1. Utilizando funciones, escribe un programa que tome dos enteros e imprima el resultado de realizar todas las operaciones bit a bit que investigó en I.2.
2. Implementa la función contarBitsEstablecidos(int num) que cuente el número de bits establecidos (1) en un número entero.
3. Escribe un programa que intercambie el valor de dos enteros a y b utilizando operadores bit a bit sin usar una variable temporal.



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 11:

Gestión dinámica de memoria y estructuras

Resumen del Curso

SEGUNDA PARTE:

1. Gestión Dinámica de Memoria

2. **Estructuras (+ Gestión dinámica de memoria):** C++ permite definir punteros a estructuras, al igual que a cualquier tipo de variable, esto conlleva a la gestión dinámica de memoria con estructuras.

3. Archivos

4. Introducción a la POO.

Contenido

1. Punteros: errores frecuentes
2. Estructuras: repaso
3. Anidamiento de una Estructura
4. Puntero a Estructura, asignación dinámica

1. Punteros: errores frecuentes

- Cuando se utilizan punteros, es difícil encontrar los errores, ya que **no se generan mensajes de error** y el programa se comporta de manera impredecible.

Sugerencias:

- Evitar utilizar punteros que apunten a variables estáticas y en especial a las variables locales. [Ejemplo](#)
- Todo puntero ha de apuntar a una dirección de memoria reservada, o si no, asignarle el valor nullptr (buena práctica) [Ejemplo](#)
- No olvidar reservar memoria (cuando sea necesario) antes de utilizar el puntero.
- No confundir la asignación de punteros con la asignación de contenidos ($p = q$; con $*p = *q$).

2. Estructuras: Repaso

- `int a;` define una variable (un espacio de memoria) que solo podrá almacenar un valor.
- `int a[5];` define un conjunto de 5 variables todas identificadas por un mismo nombre. Todos los elementos del arreglo tienen el mismo tipo de dato (inconveniente)
- Una **Estructura** (registro) permite definir un conjunto de datos identificados con el mismo nombre donde cada elemento (miembro) del conjunto puede ser de diferente tipo.
- La evolución de las estructuras dio lugar a las clases definida en programación orientada a objetos (POO) e implementadas en C++

Estructura: Definición

```
struct nombre {  
    int myNum;          // miembro (variable int)  
    string myString;   // miembro (variable string)  
};
```

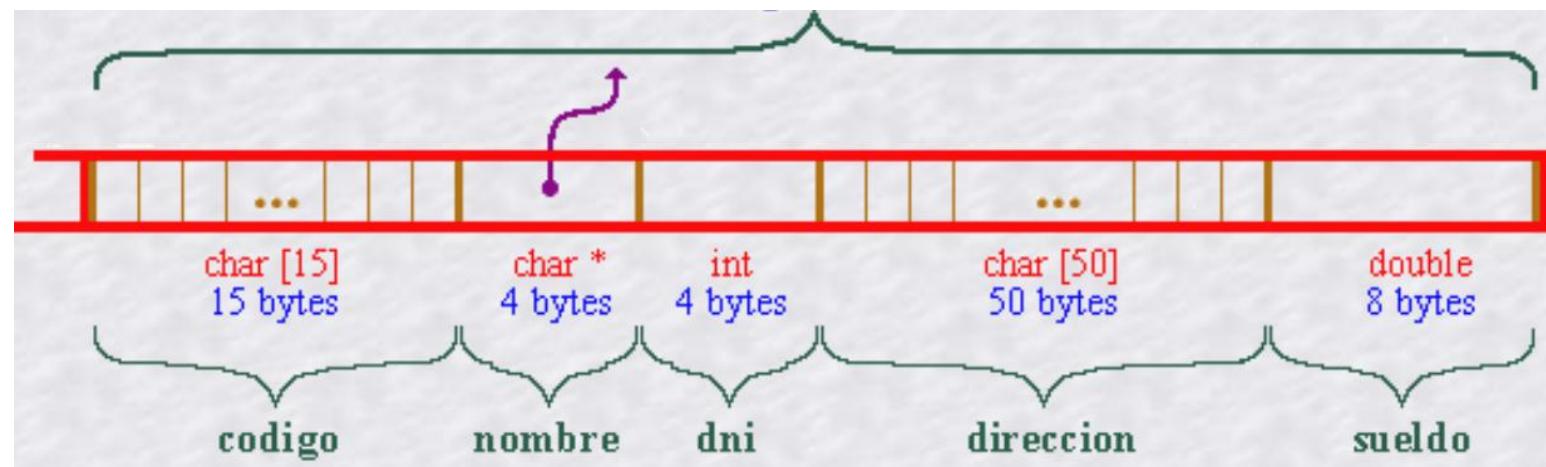
- A diferencia de los arreglos al definir una estructura **no estamos definiendo una variable directamente**, si no un "Tipo de dato". Esto quiere decir que el nombre dado a la estructura servirá para declarar variables de ese tipo

Estructura: declaración, representación interna

- Una variable de tipo struct se almacena en la pila, como se muestra a continuación:

```
struct TPersona {  
    char codigo[15];  
    char *nombre;  
    int dni;  
    char direccion[50];  
    double sueldo;  
};
```

TPersona empleado;



Estructura: Asignación de datos

- Una vez declarada la variable ésta se podrá manipular, asignando valores a sus campos o utilizando los valores de éstos en otras expresiones.

```
strcpy(empleado.codigo, "995-535262");
empleado.nombre = new char [30];
strcpy (empleado.nombre, "Juan Lopez");
empleado.dni = 82716612; //...
```

- Otra forma de asignar valores a una variable de tipo struct es al momento de declararla, esta operación se hace de la siguiente manera:

```
TPersona empleado = {"995-535262", "Juan Lopez", 82716612, "Av. ABC 123", 12345.67};
```

- La manera en que se almacenan las variables de tipo struct permite manejar las variables como una unidad, y a **diferencia de los arreglos**, se podrán **asignar los valores de todos los campos de una variable a otra en una sola operación**.

Ejemplo: Asignación de datos

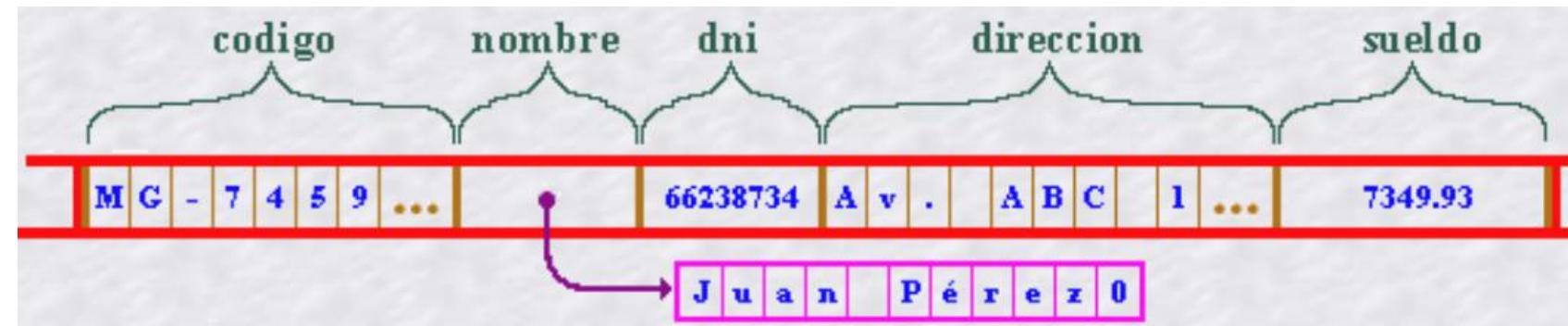
```
TPersona trabajador;  
TPersona empleado = {"995-535262", "Juan Lopez", 82716612, "Av. ABC 123", 12345.67};  
trabajador = empleado; // pasamos el contenido de la variable empleado  
                      // a la variable trabajador
```

- Aunque la operación de asignación se puede dar entre variables de tipo struct, hay que tener cuidado cuando asignamos datos a sus miembros.
- Ejemplo: Si reasignamos el campo dni de trabajador, esto no afecta al valor de dni en empleado. Sin embargo, si reasignamos el campo nombre, el cambio sí afecta a ambos. Esto ocurre porque nombre es un puntero; al realizar trabajador = empleado, se copia la dirección de memoria a la que apunta nombre en empleado hacia nombre en trabajador, haciendo que ambos apunten al mismo lugar. Por lo tanto, cualquier modificación en el contenido de nombre en trabajador también será visible en empleado.

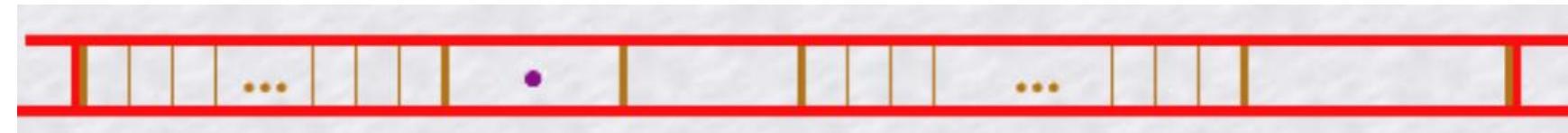
Estructura: Asignación de datos

Antes de la asignación:

empleado



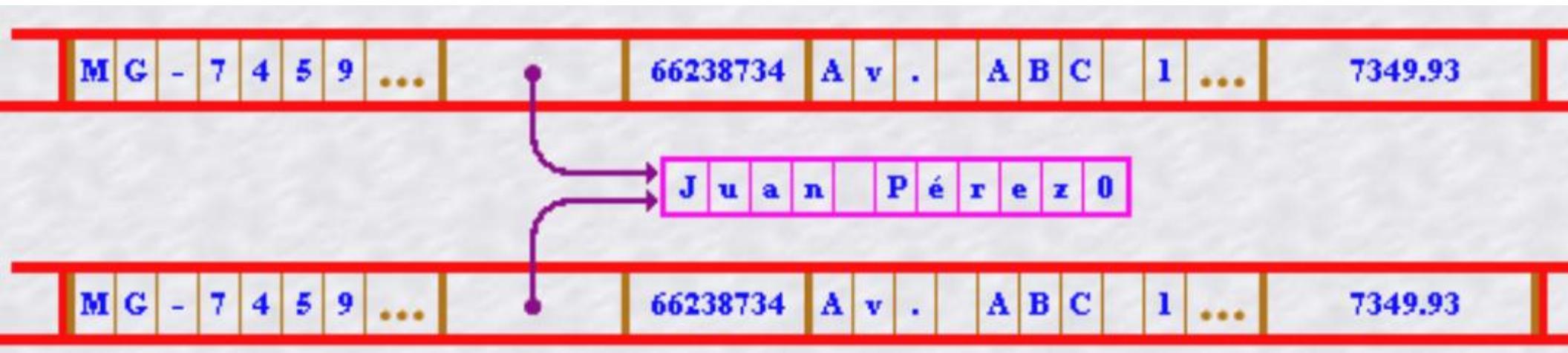
trabajador



Estructura: Asignación de datos

Luego de la asignación: trabajador = empleado;

empleado



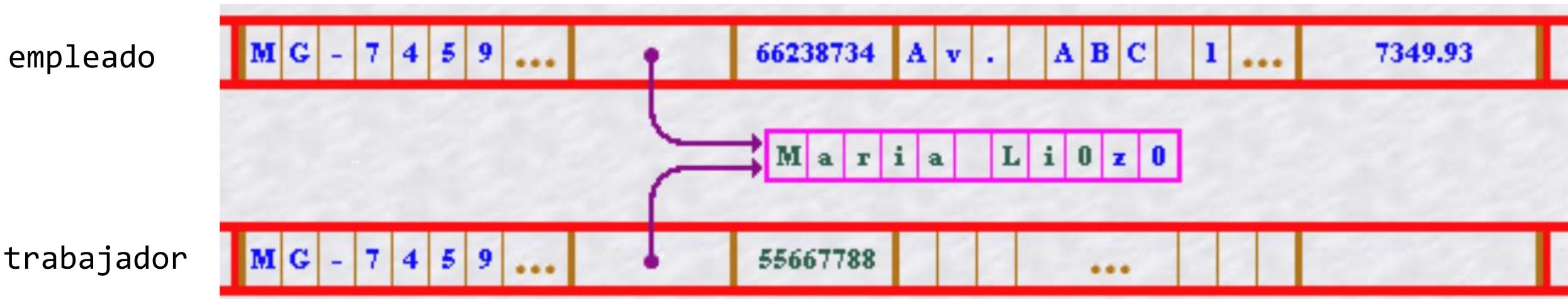
trabajador

Estructura: Asignación de datos

Luego de la asignación a sus miembros:

`trabajador.dni = 55667788;`

`strcpy (trabajador.nombre, "Maria Li");`



Advertencia: Si la estructura contiene punteros, una asignación directa entre estructuras copiará las direcciones de memoria y no los valores, por lo tanto, los cambios en un campo de la estructura afectarán a la otra.

Estructura: Asignación de datos

Otra forma de asignar datos a una variable de tipo struct es mediante funciones. **Una función puede devolver un dato de tipo struct**

El siguiente ejemplo muestra cómo:

- Definir una función que devuelve una estructura
- Utilizar punteros como miembros de una estructura
- Limpiar el buffer
- Asignación dinámica a miembros de una estructura

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

struct Persona {
    int edad;
    char* nombre;
    float altura;
}

Persona crearPersona() {
    Persona p;
    cout << "Ingrese la edad: ";
    cin >> p.edad;
    cin.ignore();
    cout << "Ingrese el nombre: ";
    char buffer[100];
    cin.getline(buffer, 100);
    p.nombre = new char[strlen(buffer) + 1];
    strcpy(p.nombre, buffer);

    cout << "Ingrese la altura: ";
    cin >> p.altura;
    return p;
}
```

...continuación

```
int main() {  
  
    Persona empleado = crearPersona();  
  
    cout << "\nDatos del empleado:" << endl;  
    cout << "Nombre: " << empleado.nombre << endl;  
    cout << "Edad: " << empleado.edad << endl;  
    cout << "Altura: " << empleado.altura << endl;  
  
    delete[] empleado.nombre;  
  
    return 0;  
}
```

Comparación de datos entre estructuras

- A pesar de que podemos asignar las estructuras de datos directamente, **no se pueden comparar de forma directa**. Debido a la forma cómo se representa una estructura en memoria.
- Al comparar estructuras, se debe hacer esta operación miembro a miembro y no como una unidad.

3. Estructuras anidadas

Un tipo de dato struct se puede emplear como tipo de dato para definir uno o más miembros (campos) en otra estructura. El ejemplo siguiente muestra esta propiedad:

```
struct TFecha {  
    int dd;      // día  
    int mm;      // mes  
    int aa;      // año  
};  
struct TEmpleado{  
    int dni;  
    char *nombre;  
    char *direccion;  
    TFecha fechaDeNacimiento; // miembro de tipo struct  
    char *cargo;  
    TFecha fechaDeIngreso; // miembro de tipo struct  
};
```

Estructuras anidadas

Para manejar los campos en la estructura anidada se sigue la misma lógica que en el caso de una estructura simple. Considerando el ejemplo anterior:

```
int main() {  
    TEmpleado empleado;  
    empleado.fechaDeNacimiento.dd = 27;  
    empleado.fechaDeNacimiento.mm = 3;  
    empleado.fechaDeNacimiento.aa = 1989;  
}
```

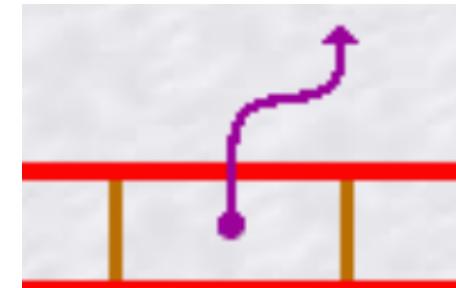
4. Punteros a estructuras, asignación dinámica

- Podemos definir un puntero a una estructura, como en cualquier variable.
- El siguiente ejemplo muestra una estructura sencilla, y a partir de ella veremos en detalle el manejo de la variable referenciada.

```
struct TPersona {  
    int dni;  
    char *nombre;  
    double sueldo;  
};  
  
int main() {  
    TPersona *empleado; // declaración (puntero a variable struct)  
    empleado = new Tpersona; // asignación de memoria  
                                //se crea la variable referenciada  
}
```

Puntero a estructura: representación

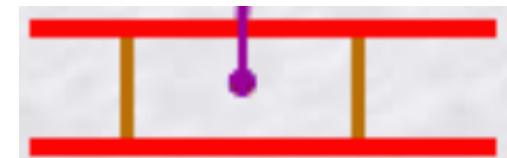
- TPersona *empleado;
crea el espacio en la pila solo para el puntero
- empleado = **new** TPersona;
Esta instrucción crea un bloque de memoria en el montón (heap) suficiente para almacenar una estructura del tipo TPersona y asigna la dirección de este bloque al puntero empleado. De esta forma, empleado puede referenciar y manipular la estructura en la memoria dinámica.



empleado



*empleado



empleado

Acceso a los miembros de la variable referenciada

- Cuando se hace mención al miembro de una variable de tipo struct el compilador reconoce al nombre de la variable, al punto y al nombre del miembro como el identificador.
- **Error común** : escribir `*empleado.dni` es un error porque el campo dni no es un puntero, es una variable de tipo int. El operador **. tiene mayor precedencia** que el operador *****
- La manera correcta de llegar al campo dni es: `(*empleado).dni` , de ese modo el operador ***** se aplica sobre el puntero y no sobre el campo.
- **Forma alternativa:** usar el operador flecha `->`, para llegar al campo de la variable referenciada. Ejemplo: `empleado -> dni`

```
#include <iostream>
#include <string>
using namespace std;
struct Empleado{
    string nombre;
    string n_telef;
};
void trabajadores(Empleado *); // prototipo de funcion
void mostrar(Empleado *); // prototipo de funcion
int main(){
    char key;
    Empleado *recPoint;
    cout << "Desea crear un nuevo registro? (y/n): ";
    key = cin.get();
    if (key == 'y') {
        key = cin.get(); //leer '\n' en la entrada almacenada en el búfer <>cin.ignore()
        recPoint = new Empleado; //ASIGNACIÓN DINÁMICA ...liberar??
        trabajadores(recPoint);
        mostrar(recPoint);
    }
    else{
        cout << "\nNo se creo ningun registro.";
        return 0;
    }
}
```

...continuación

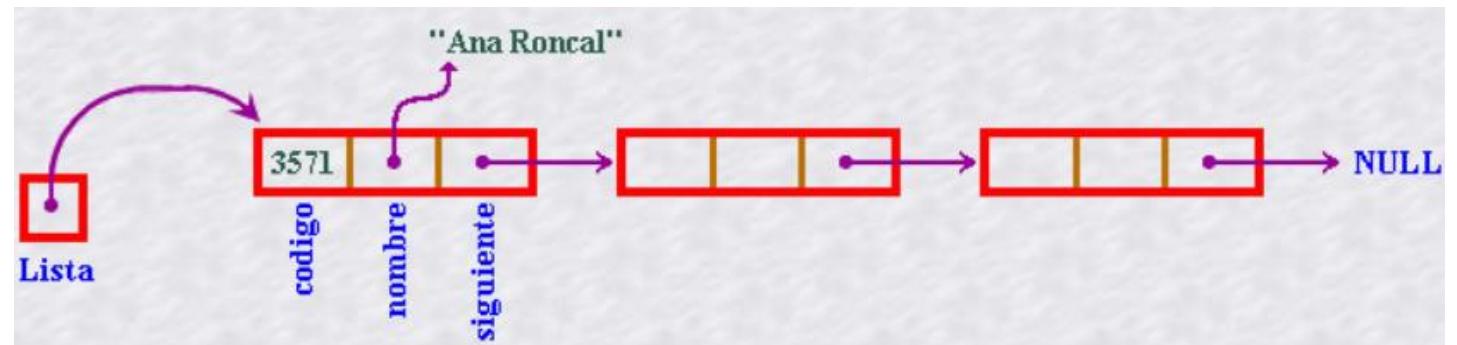
```
// Ingresar un nombre y un numero de telefono
void trabajadores(Empleado *record) {
    cout << "Ingrese un nombre: ";
    getline(cin,record->nombre);
    cout << "Ingrese el numero de telefono: ";
    getline(cin,record->n_telef);
}

// Mostrar el contenido del registro
void mostrar(Empleado *cont){
    cout << "\nContenidos del registro creado:" << "\nnombre: " << cont->nombre
    << "\nNumero de Telefono: " << cont->n_telef << endl;
}
```

Estructuras autoreferenciadas

- Una estructura autoreferenciada es una variable de tipo struct en la que uno o más de sus miembros (campos) son punteros del mismo tipo que la estructura que lo define.
- Bajo este concepto, una variable definida así puede enlazarse (apuntar), a través de estos miembros, a otra u otras variables del mismo tipo, generando las Listas enlazadas (ligadas), Árboles, etc.

```
struct NodoLSL {  
    int codigo;  
    char *nombre;  
    NodoLSL *siguiente;  
};
```





UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente: Américo Chulluncuy Reynoso**

2025-II

Sesión 13:

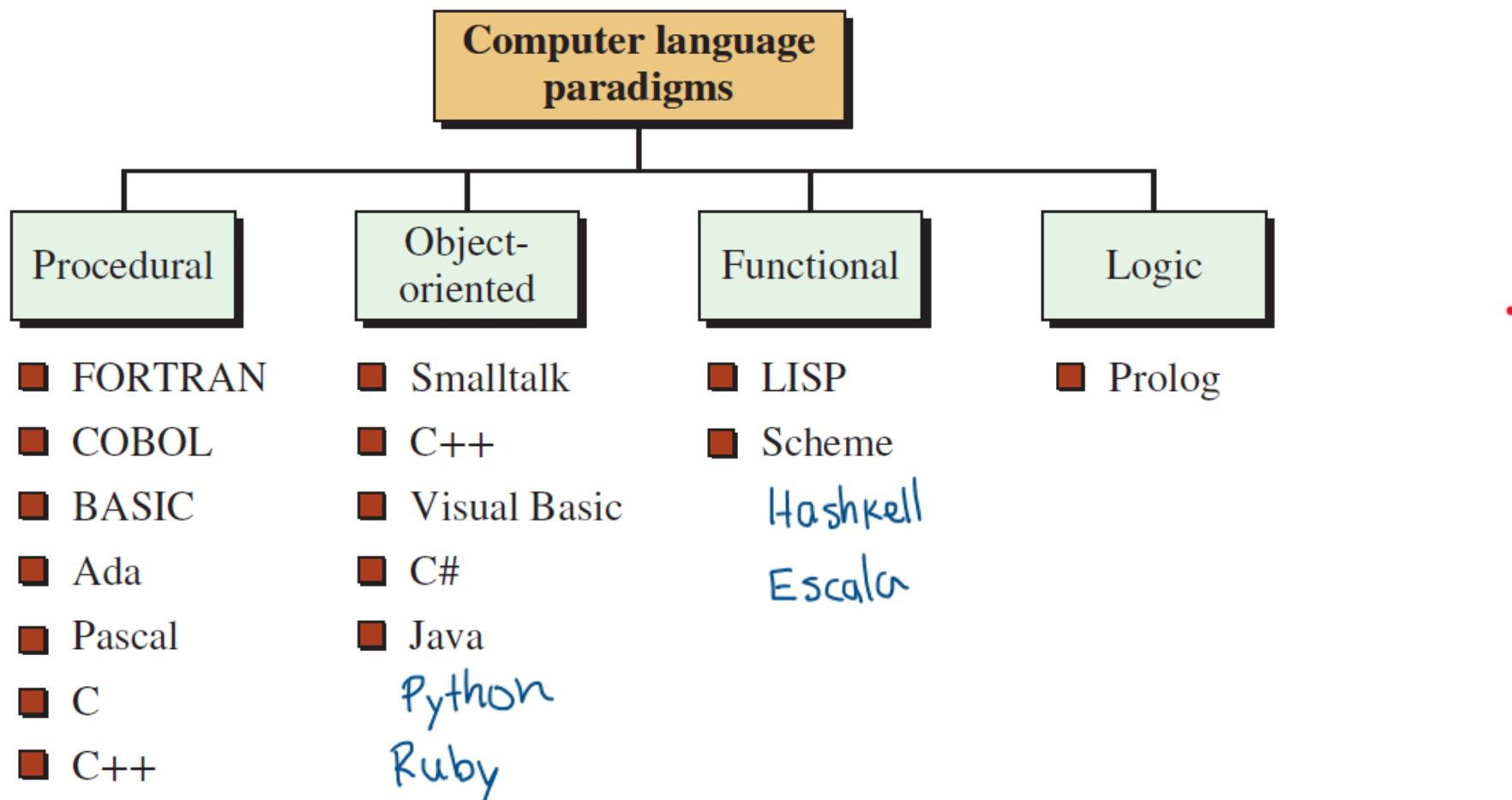
Clases

Contenido

1. Introducción
2. Clases y objetos
3. Constructores y destructores
4. Puntero this:
5. Uso del puntero this en un constructor
6. Tarea

1. Introducción

- Los lenguajes informáticos se pueden clasificar según el enfoque que utilizan para resolver un problema.
- Un **paradigma** es un modelo para describir cómo un programa maneja los datos.

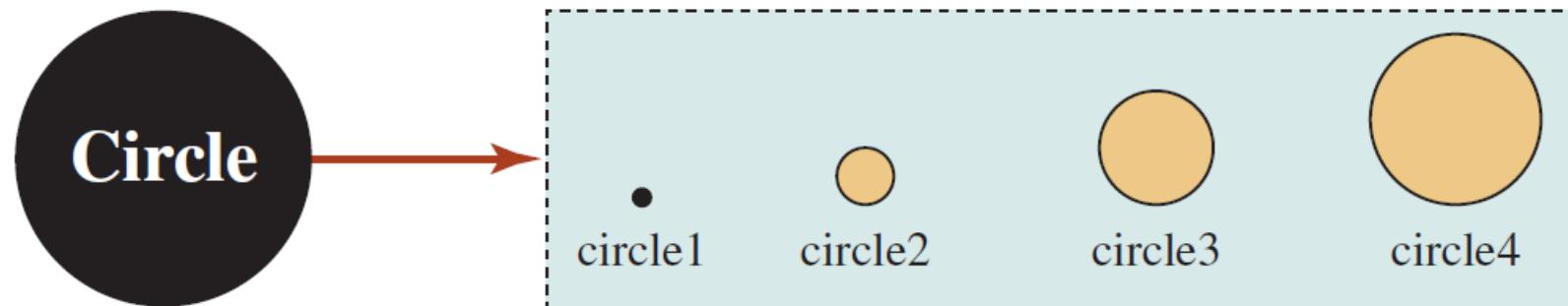


Paradigmas de lenguaje

- **Programación procedural:** El programa se estructura en torno a procedimientos o funciones que manipulan datos.
- **Programación Orientada a Objetos (POO):** Gira en torno al concepto de objetos, que son instancias de clases. Los objetos interactúan entre sí mediante métodos, este enfoque facilita la reutilización de código, modularidad y el mantenimiento.
- **Programación funcional:** trata la computación como la evaluación de funciones matemáticas y evita cambios de estado y datos mutables.
- **Programación lógica:** Utiliza un conjunto de hechos y un conjunto de reglas para responder consultas. Se basa en la lógica formal.

2. Clases y Objetos

- Un tipo es una abstracción; una instancia de ese tipo es una entidad concreta. Ejemplo:



- La relación entre un tipo y sus instancias es una **relación de uno a muchos**.
- Cualquier instancia (real) tiene:
Atributos: cualquier característica que nos interese
Comportamientos: operación que la instancia puede realizar sobre sí misma.

Atributos y comportamientos

Ejemplo: Atributos

- Instancia de Circle: radio, perímetro, área.
- Instancia de Empleado: nombre, dirección, salario.
- Instancia de Alumno: ?

Ejemplo: Comportamientos

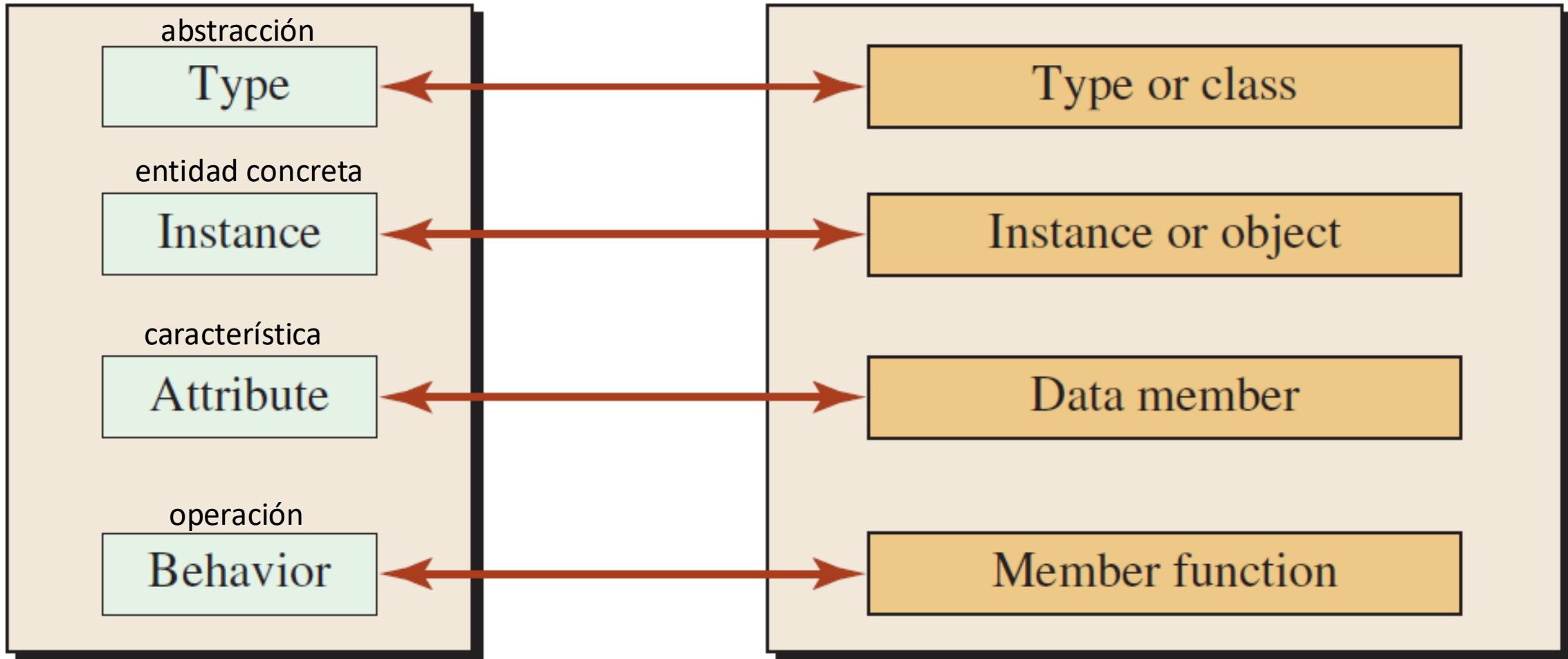
- Instancia de Circle: puede dar su radio, su perímetro y su área.
- Instancia de Empleado: esta puede dar su nombre, dirección y salario.
- Instancia de Alumno: ?

Clases y objetos en programas

- En POO, creamos un tipo (definido por el usuario) utilizando **class**. Una instancia de una clase se denomina **objeto**.
- En POO, los atributos y comportamientos de un objeto son representados como miembros de datos y funciones miembros.
 - ✓ Un **miembro de dato** de un objeto es una variable cuyo valor representa un atributo. Por ejemplo, en el caso del círculo, **double radio**;
 - ✓ Una **función miembro** (método) simula uno de los comportamientos de un objeto. Por ejemplo, podemos escribir una función que permita a un círculo dar su radio.

Comparación de términos

Realidad



POO

Clases

- Una clase es una colección de un número fijo de componentes llamados **miembros** de la clase.
- **Sintaxis para definir una clase**

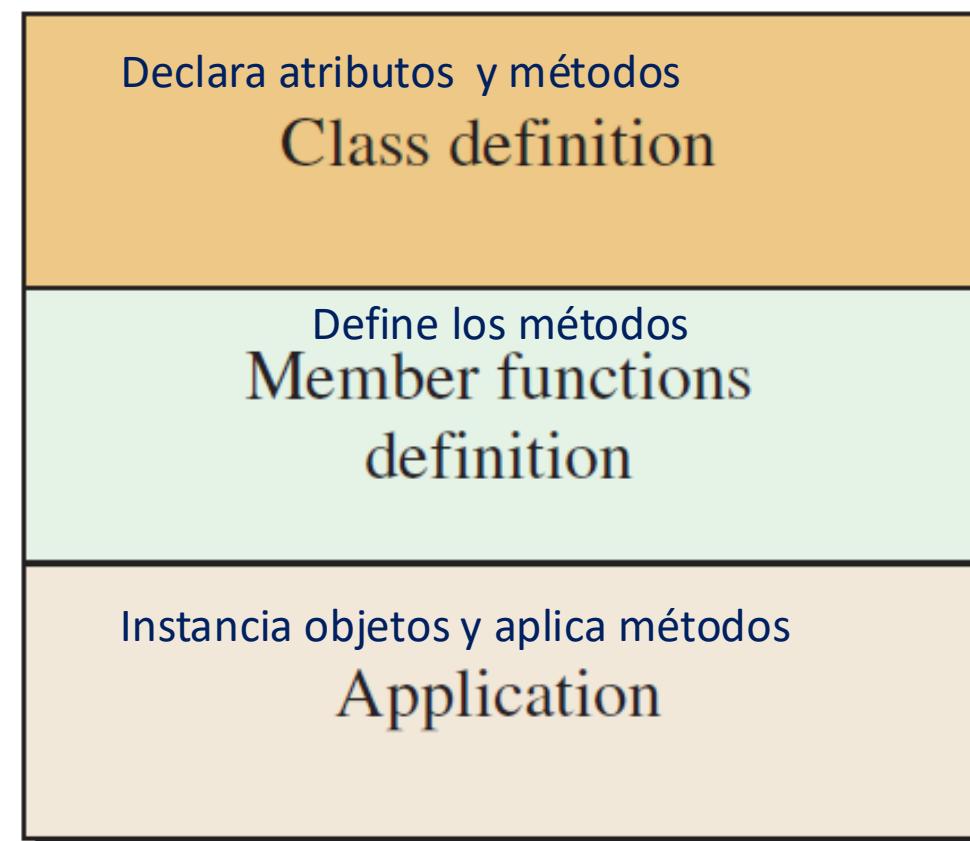
```
class IdentificadorClase{  
    ListaMiembrosClase  
};
```

- **ListaMiembrosClase** consta de declaraciones de variables y/o funciones, es decir, un miembro de una clase puede ser una variable (para almacenar datos) o una función (para manipular datos).

Programas en el paradigma orientado a objetos

- A diferencia de la programación procedural, en POO necesitamos **tres secciones**:
 - ✓ **Sección 1: Definición de clase,**
 - ✓ **Sección 2: Definición de función miembro**
 - ✓ **Sección 3: Aplicación**
- En algunos casos cada sección es diseñada y almacenada por diferentes entidades.

Por ejemplo en las clases string y fstream, las 2 primeras secciones son creadas y almacenadas en la biblioteca estándar de C++.



Sección 1: definición de clase

- Consta de tres partes: encabezado, cuerpo y punto y coma. Ejemplo:

```
class Circle {  
private: //modificador de acceso (de grupo)  
    double radius; //declaración de miembros de datos o variables (Atributos)  
                  //variables integradas u otros tipos de clases previamente definidas  
public:  
    double getRadius () const; //declaración de funciones miembro (Métodos)  
    double getArea () const; //const, significa que la función no puede modificar los  
    double getPerimeter () const; //miembros de datos de una variable de tipo Circle  
    void setRadius (double value);  
};
```

- El **modificador de acceso** determina cómo se puede acceder a una clase. Por defecto es **private** (**no se puede acceder a los datos y funciones miembros para recuperarlos o modificarlos**)

Modificadores de acceso

- Para evitar esta limitación, C++ define tres modificadores de acceso:

Modifier	Access from same class	Access from subclass	Access from anywhere
private	Yes	No	No
protected	Yes	Yes	No
public	Yes	Yes	Yes

- Los **miembros de datos** normalmente se configuran como **private**. Se debe acceder a ellos a través de las funciones miembros.
- Las **funciones miembros** normalmente deben establecerse como **public**.

Sección 2: Definición de funciones miembro

- Cada función miembro necesita una definición usual, **con dos diferencias**:
(i) const y (ii) el nombre de la función que debe calificarse con el nombre de la clase seguido de un símbolo de alcance de clase ::

```
double Circle :: getRadius () const { //la función no puede modificar el objeto
    return radius;
}
double Circle :: getArea () const {
    const double PI = 3.14;
    return (PI * radius * radius);
}
double Circle :: getPerimeter () const {
    const double PI = 3.14;
    return (2 * PI * radius);
}
void Circle :: setRadius (double value) {
    radius = value;
}
```

Funciones en línea

- El tiempo de ejecución asociado a una llamada de función puede ser mayor que ejecutar directamente el código contenido en la función. Para mejorar el rendimiento, se puede declarar una función como función en línea (inline), lo que **sugiere** al compilador reemplazar la llamada por el cuerpo de la función en tiempo de compilación.
- Una función en línea implícita ocurre cuando se define directamente dentro de la declaración de la clase. Aunque es válida no se recomienda porque:
 - (i) Dificulta la legibilidad de código.
 - (ii) Puede violar el principio de encapsulamiento (*?!)
- Una función en línea explícita se define utilizando la palabra clave `inline` antes de su definición, generalmente fuera de la clase. Esto permite mantener la separación entre interfaz e implementación, preservando así una mejor organización del código.

Sección 3: Aplicación

- Utiliza la definición de clase y la definición de función miembro. **Crea instancias de objetos de la clase y aplica las funciones miembros en esos objetos**

```
Circle circle1;
```

```
circle1.setRadius (10.0);
```

```
cout << "Radio: " << circle1.getRadius() << endl;
cout << "Area: " << circle1.getArea() << endl;
cout << "Perímetro: " << circle1.getPerimeter() << endl;
```

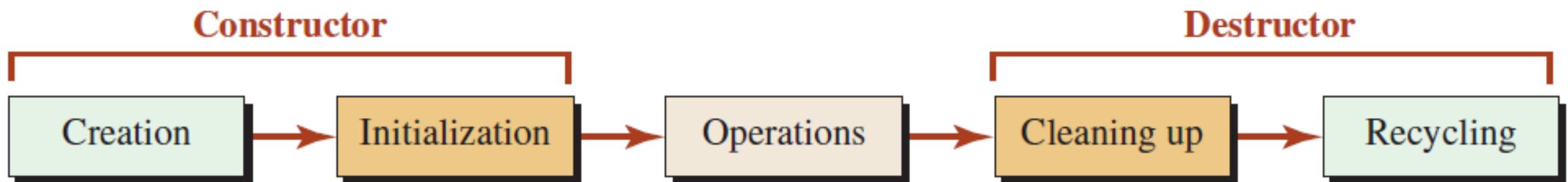
- **Ejemplo1**

3. Constructores y destructores

- Para que un objeto realice operaciones sobre sí mismo, primero debe crearse. La **creación ocurre al invocar el constructor**. Durante la construcción:
 - ✓ Primero se ejecuta la lista de inicialización (antes del cuerpo del constructor).
 - ✓ Luego se ejecuta el cuerpo del constructor para realizar otras tareas opcionales.
- Cuando un objeto ya no es necesario, debe liberarse y sus recursos deben limpiarse. La **limpieza se realiza automáticamente mediante una función miembro especial llamada destructor**. El destructor se ejecuta **cuando el objeto sale de su alcance**. La memoria del programa se recicla completamente al finalizar la ejecución, pero **los objetos deben limpiarse antes de eso** para evitar pérdidas de memoria (memory leaks) o dejar recursos sin liberar.

Constructores y destructores

- Por lo tanto, un **objeto pasa por cinco pasos**. Es **creado e inicializado** por una función miembro especial llamada constructor. **Aplica algunas operaciones** solicitadas por la aplicación sobre sí misma. Es **limpiado y reciclado** por otra función miembro especial llamada destructor.



Constructor

- Es una función miembro que **crea un objeto cuando se llama e inicializa los datos miembros de un objeto cuando se ejecuta.**
- Un constructor **tiene dos características:** (1) **no tiene valor de retorno** y su (2) **nombre es el mismo que el nombre de la clase.**
- Podemos tener **tres tipos de constructores en una clase:** constructores de parámetros, constructores predeterminados y constructores de copia.
- **Declaración:**

```
class Circle{
    public:
        Circle (double radius); //Constructor de parámetros
        Circle (); //Constructor predeterminado
        Circle (const Circle& circle); //Constructor de copia
}
```

Tipos de constructores

- Un **constructor de parámetros** inicializa los miembros de datos de cada instancia con valores específicos. Podemos tener varios constructores de parámetros, cada uno con un valor diferente (sobrecarga).
- El **constructor predeterminado** no tiene parámetros. Útil para crear objetos donde cada miembro de datos es establecido en algunos valores literales o **valores predeterminados**. No lo podemos sobrecargar.
- Un **constructor de copia**, copia los valores de los miembros de datos del objeto dado al nuevo objeto recién creado. **Tiene solo un parámetro por referencia** (al objeto fuente). No lo podemos sobrecargar porque la lista de parámetros es fija. No podemos cambiar el objeto fuente

Constructor: Definición

```
// Definición de constructor de parametros
Circle :: Circle (double rds)
: radius (rds) // lista de inicialización de
                 //miembros antes del cuerpo
{
// ...
}

// Definición de constructor predeterminado
Circle :: Circle ()
: radius (1.0) // Lista de inicialización.
{
// ...
}

// Definición de constructor copia
Circle :: Circle (const Circle& cr)
: radius (cr.radius) // lista de inicialización
{
// ...
}
```

- Note que un constructor **puede tener una lista de inicialización después del encabezado para inicializar los miembros de datos.**
- Si necesitamos inicializar más de un miembro de dato, **la inicialización de cada miembro de datos debe estar separada por una coma de otros miembros de datos.**
- El nombre de cada parámetro lo determina el programador, pero el del miembro de datos debe ser el mismo
- **Podemos usar el cuerpo del constructor para inicializar miembros de datos complejos, para validar un parámetro, para abrir archivos si es necesario o incluso imprimir un mensaje para verificar que se llamó al constructor.**

Destructor

- Es una función miembro que **limpia y destruye un objeto**. Tiene **dos características especiales**.
 1. Tiene el mismo nombre de la clase precedido por ~
 2. No puede tener un valor de retorno
- El **sistema llamará y ejecutará automáticamente un destructor cuando el objeto instanciado de la clase salga del alcance**.
- La limpieza es importante si lo construido ha llamado recursos como archivos. Una vez finalizado el programa, la memoria asignada se recicla.
- Un destructor no puede aceptar argumentos, lo que significa que no puede sobrecargarse.

Destructor: declaración y definición

- Declaración:

```
class Circle {  
public:  
    ~Circle ();  
}
```

- Definición:

```
Circle :: ~Circle (){  
// ...  
}
```

- Llamada

**Parameter
constructor**

```
Circle circle1 (5.1);
```

**Default
constructor**

```
Circle circle2;
```

**Copy
constructor**

```
Circle circle3 (aCircle);
```

Destructor

Called by system

Ejemplo: uso de constructores y destructor

Creación de variables vs objetos de clases

La siguiente tabla compara la creación de variables de tipos integrados con la creación de objetos de tipos clase.

Member	Class type	Built-in type
Parameter constructor	Circle circle1 (10.0);	double x1 = 10.0;
Default constructor	Circle circle2;	double x2;
Copy constructor	Circle circle3 (circle1);	None
Destructor	No call	No call

Funciones miembro requeridas por una clase

- G1. **Constructor de parámetros y el constructor predeterminado.** Debemos tener al menos uno de estos constructores. Si proporcionamos cualquiera de ellos, el sistema no nos proporciona ninguno. Si no proporcionamos ninguno de ellos, el sistema proporciona un constructor predeterminado, que inicializa cada miembro con lo que encuentran en el sistema (ver [Ejemplo 1](#)).
- G2. **Constructor de copias.** Una clase debe tener **un único constructor de copia**, pero si no proporcionamos uno, el sistema nos proporciona uno. La mayoría de las veces **es mejor crear nuestro propio constructor de copias**.
- G3. **Destructor.** Una clase debe tener **un solo destructor**, pero si no proporcionamos uno, el sistema nos proporciona uno. **Es mejor crear nuestro propio destructor.**

Ejemplo 2

En el [Ejemplo 2](#), volvemos a considerar el Ejemplo 1 de la clase Circle ahora con los constructores y el destructor proporcionados. Incluimos un mensaje en el cuerpo de cada constructor y destructor para mostrar cuándo se llaman

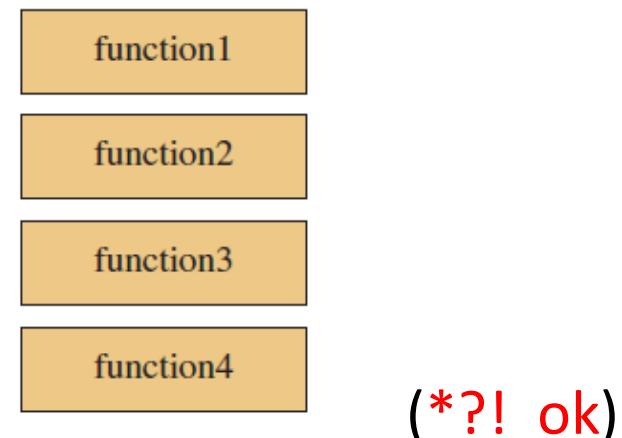
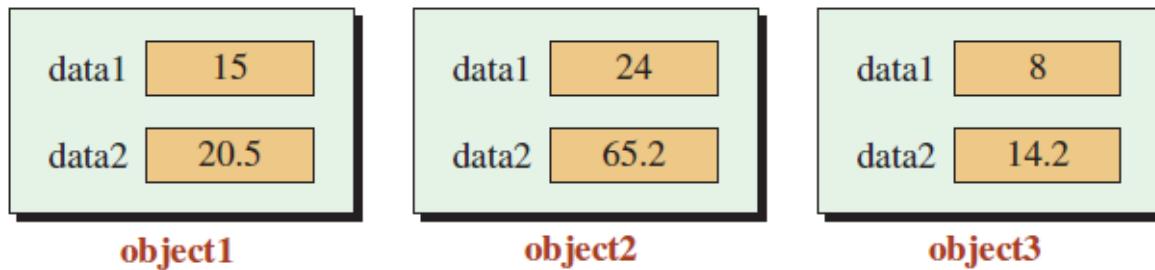
```
class Circle {  
    private:  
        double radius; //declaración de miembros de datos  
(variables)  
    public:  
        Circle(double radius); // Constructor de parámetro  
        Circle(); //Constructor predeterminado  
        ~Circle(); //Destructor  
        Circle(const Circle& circle); // constructor copia  
        void setRadius (double value); //Mutador  
        double getRadius () const; // Accesor  
        double getArea () const; //Accesor  
        double getPerimeter () const; // Accesor  
};
```

Ejemplo 3: Generando números aleatorios

- En este caso solo necesitamos definir el constructor de parámetros (el sistema no crea un constructor predeterminado) porque los constructores predeterminados y de copia no tienen sentido.
- `delete` evita que el sistema proporcione un constructor de copia (C++11)
- Solo dos de los tres miembros de datos (`low` y `high`) se inicializan en la lista de inicialización del constructor. El tercer miembro de datos (`valor`) se calcula en el cuerpo del constructor.
- La única función miembro accesora es la función `print`, que imprime el valor del número aleatorio creado. No tenemos funciones miembros mutadoras porque no queremos cambiar el número aleatorio creado.

Miembros de un objeto

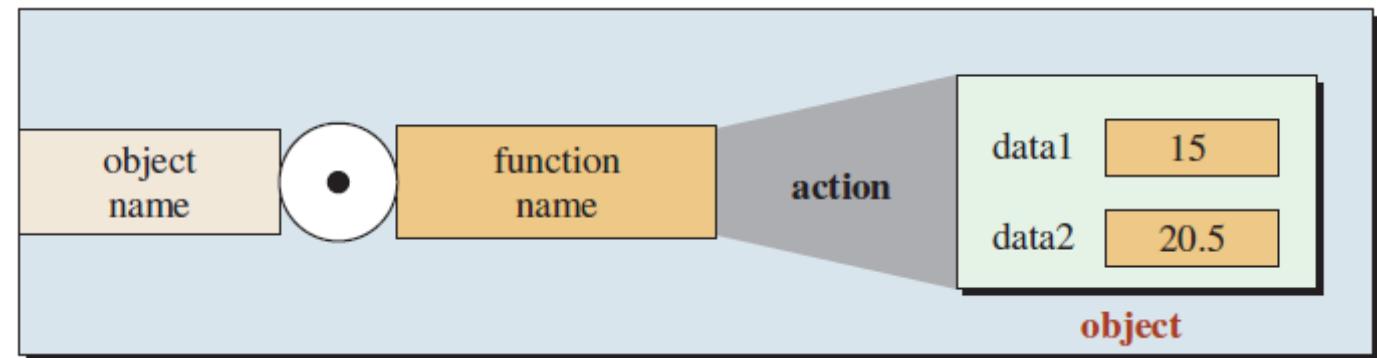
- **Miembros de datos:** define los atributos del objeto. Cada objeto debe **encapsular** (**se asignan regiones de memoria separadas para cada objeto**) el conjunto de miembros de datos definidos en la clase. Otros objetos no pueden acceder a ellos. Normalmente es privado
- **Funciones miembros:** define uno de los comportamientos que se pueden aplicar en los miembros de datos de un objeto. **Solo hay una copia de cada función miembro** del objeto en la memoria y debe compartirse por todas las instancias. Normalmente es público.



Selectores de funciones miembros

- Una aplicación puede llamar a una función miembro de instancia para operar en ella. En POO esta llamada debe realizarse a través de la instancia (objeto)
- La aplicación primero debe construir una instancia y luego dejar que la instancia llame a la función miembro. El lenguaje C++ define dos operadores, llamados operadores selectores de miembros, para este propósito.

Operator	Expression
.	object.member
->	pointer -> member



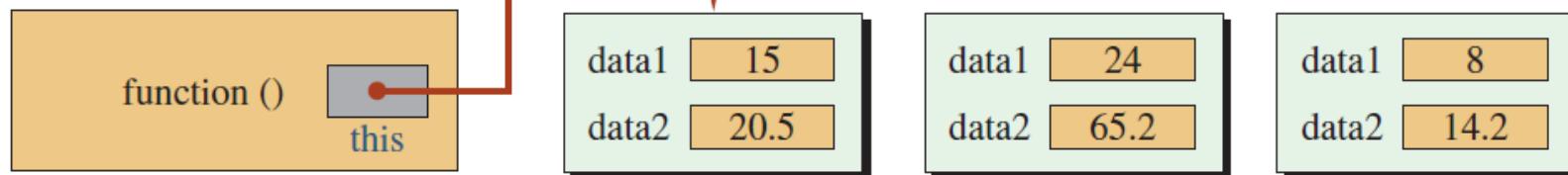
- Ver Ejemplos 2 y 3

Puntero this

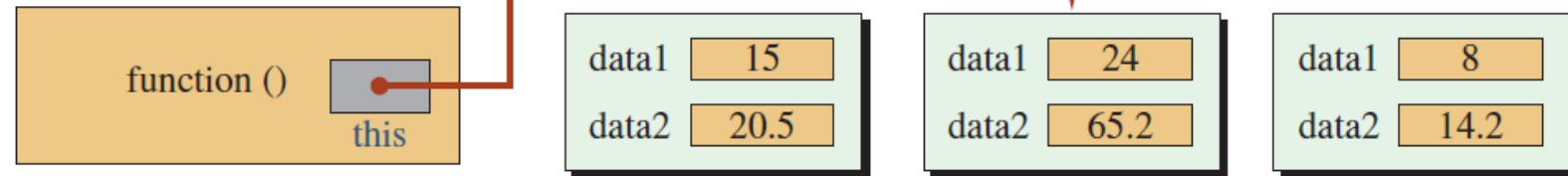
- Dado que solo hay una copia de una función miembro, para bloquear una función cuando se está utilizando y desbloquearla cuando la función finaliza, C++ agrega un puntero a cada función miembro.
- Mientras usamos el selector de miembros de punto, **el compilador lo cambia a un selector de miembros de puntero**, en el que **cada función miembro tiene un puntero oculto llamado puntero this**.
- La función es empleada por el objeto al que apunta este puntero en ese momento (el código de función se aplica a los miembros de datos del objeto al que apunta el puntero this)

Puntero this

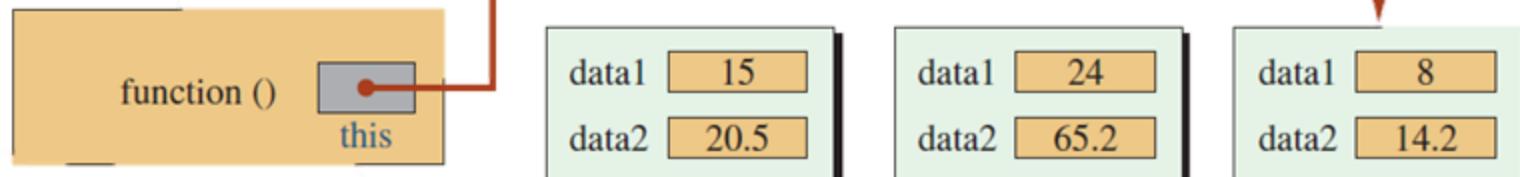
At time 1



At time 2



At time 3



Bloqueo y desbloqueo de una función para un objeto particular

Puntero this: uso implícito

- ¿Cómo obtiene una función miembro el puntero this?

El compilador lo agrega como parámetro a la función miembro de la instancia como se muestra a continuación:

```
// Lo que el usuario escribe
double getRadius () const{
    return radius;
}

// Cambio del compilador
double getRadius (Circle* this) const{
    return (*this).radius;
}
```

circle1.getRadius();

this = &circle1;
getRadius (this);

- **Recordemos:** this -> radius es equivalente (*this).radius

Puntero this: uso explícito

- Podemos usar el puntero this en nuestro programa para **hacer referencia a un miembro de datos** en lugar de usar el miembro de datos en sí, y podemos usar el nombre del miembro de datos como parámetro.
- De esta forma, no tenemos que utilizar nombres abreviados

```
// sin usar el puntero this
void Circle :: setRadius (double rds){
    radius = rds;
}

// usando el puntero this
void Circle :: setRadius (double radius) {
    this -> radius = radius;
}
```

Puntero this: uso explícito

- **objeto anfitrión** (host) Es un **objeto sobre el que está operando la función miembro en un momento dado.**
- El puntero this no se puede utilizar en la lista de inicialización de un constructor porque en ese momento el objeto anfitrión no ha sido construido.
- Sin embargo, se puede utilizar en el cuerpo del constructor si es necesario.
- El puntero this es un unique_pointer (Smart pointers)

5. Uso del puntero this en un constructor

- this apunta al objeto en el que se está trabajando actualmente. Dentro de sus funciones miembro, se utiliza para hacer referencia a los miembros y métodos del objeto actual. En general, todas las funciones miembros pueden acceder a este puntero, que el compilador crea automáticamente.
- El constructor de una clase es un lugar donde se utiliza con frecuencia el puntero this. Cuando se crea un objeto de la clase, se invoca el constructor.
- Se utiliza para configurar requisitos adicionales e inicializar los miembros de datos del objeto. Es útil para distinguir entre los parámetros del constructor y los miembros de datos del objeto.

Ejemplo

```
#include <iostream>
using namespace std;
class MyClass {
    private:
        int value;
    public:
        MyClass(int value) { //para diferenciar entre parámetro local y variable miembro
            this->value = value; // Inicializamos la variable miembro usando el puntero 'this'
        }
        void printValue() { //para acceder a la variable miembro e imprimir su valor
            cout << "Value: " << this->value << endl;
        }
};

int main() {
    MyClass obj(10);
    obj.printValue();
    return 0;
}
```

Adicional:

Función miembro de acceso

- Una función miembro de acceso obtiene información del objeto anfitrión(actual) pero no cambia el estado del objeto, es decir, convierte el objeto anfitrión en un objeto de sólo lectura.
- Para garantizar que una función miembro de acceso no cambie el estado del objeto, **debemos agregar el calificador `const`** al final del encabezado de la función (tanto en la declaración como en la definición)

```
//el calificador const hace que el objeto anfitrión sea solo de lectura
double getRadius () const;
double getPerimeter () const;
double getArea() const;
```

Función miembro de acceso

- Una función miembro de acceso no tiene que devolver el valor de un miembro de datos; se puede utilizar para generar un valor, siempre que no haya cambios en el estado del objeto. Ejemplo, podemos tener una función de salida que imprima el valor del radio, el perímetro y el área de un objeto de la clase Circle sin valor de retorno como se muestra a continuación.

```
void Circle :: print () const {  
    cout << "Radio: " << radius << endl;  
    cout << "Perimetro: " << 2 * radius * 3.14 << endl;  
    cout << "Area: " << radius * radius * 3.14 << endl;  
}
```

Función miembro mutador

- Son funciones que puedan cambiar el estado de sus objetos anfitriones. No debe tener el calificador const. Los constructores y destructores pueden considerarse funciones miembros mutadoras (inicializan o limpian objetos)
- Por ejemplo, si creamos una clase que representa una cuenta bancaria, el miembro de datos que representa el saldo cambia con el tiempo (con cada depósito y retiro).

```
//no es necesario const para un mutador
void setRadius (double rds);

void Circle :: input(){// puede no tener parámetros
    cout << "Ingrese el radio del objeto Circle: ";
    cin >> radius;
}
```

Invariantes de clase

- Es una o más condiciones que debemos imponer a algunos o todos los miembros de datos de instancia de una clase y que debemos hacer cumplir a través de funciones miembro de instancia.
- Ejemplo: en nuestra clase Circle, el radio debe ser un valor positivo. Aplicamos la invariante de una clase a través de funciones miembro de datos que crean objetos (constructores de parámetros) o funciones miembros mutadoras que cambian el valor de un miembro de datos.
- El [Ejemplo 4](#) muestra cómo cambiar el constructor de parámetros de una clase Rectangle para garantizar la invariante de la clase.

Miembros estáticos

- Es un miembro de datos que pertenece a todas las instancias; también pertenece a la clase misma y sus declaraciones deben incluirse en la definición de la clase. Los miembros estáticos usan la palabra clave static.
- Declaración:

```
class Rectangle{
    private:
        static int count; // miembro de datos estático
    public:
}
```

Miembros estáticos

- Un miembro de datos de instancia normalmente se inicializa en un constructor, pero un miembro de datos estáticos no pertenece a ninguna instancia, lo que significa que no se puede inicializar en un constructor.
- Un miembro de datos estáticos debe inicializarse después de la definición de clase. Esto significa que debe inicializarse en un área global del programa. Debemos demostrar que pertenece a la clase agregando el nombre de la clase y el operador de alcance de la clase (::) a la definición, pero no se debe agregar el calificador estático.
- Inicialización:
`int Rectangle :: count = 0;`

Miembros estáticos

- Una función miembro estática puede acceder al miembro de datos estáticos a través de un objeto y también a través del nombre de la clase cuando no existe ningún objeto. Una función miembro estática no tiene ningún objeto host porque no está asociada con ninguna instancia.
- Declaración:

```
//se declara dentro de la clase, usa la palabra static,  
class Rectangle {  
    private:  
        static int count; // miembro de datos estático  
    public:  
        static int getCount(); // función miembro estático  
}
```

Miembros estáticos

- Definición:

```
//debe definirse fuera de la clase  
//== a la def de una función miembro de instancia  
//No usamos static. Consultar la declaración para saber el tipo  
int Rectangle :: getCount() {  
    return count;  
}
```

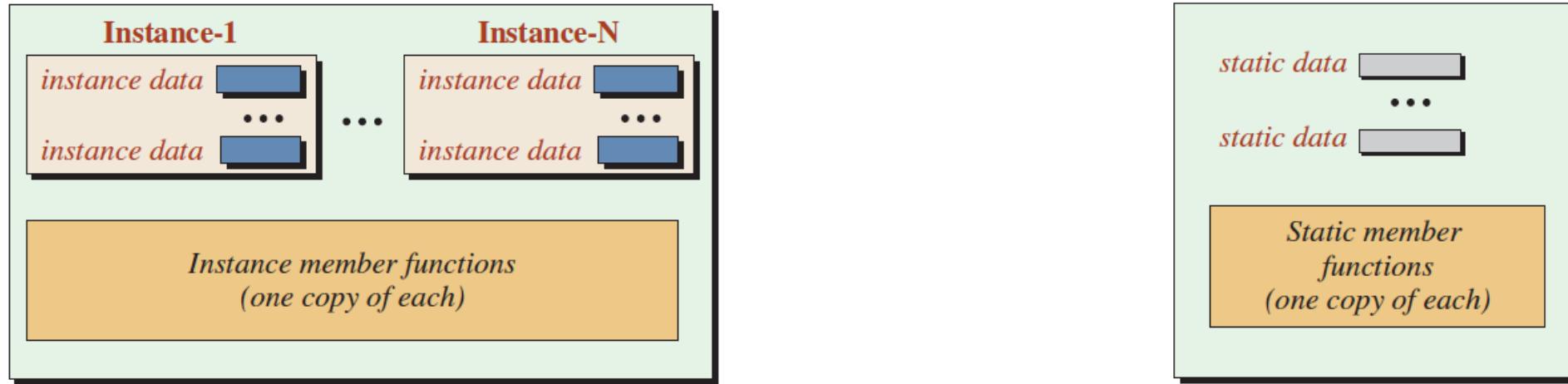
- Llamada

```
rect.getCount(); // a través de una instancia  
Rectangle :: getCount(); // a través de la clase
```

Miembros estáticos

- **Advertencia:** No podemos usar una función miembro estática para acceder a un miembro de datos de instancia porque una función miembro estática **no tiene el puntero this oculto**, que define la instancia a la que se debe hacer referencia.
- Por otro lado, se puede utilizar una función miembro de instancia para acceder a miembros de datos estáticos (no se utiliza el puntero this),
- Una buena práctica es utilizar funciones de miembros de instancia para acceder a miembros de datos de instancia y funciones de miembros estáticos para acceder a miembros de datos estáticos.

Miembros estáticos



- El [Ejemplo 5](#), muestra un programa con un miembro de datos estáticos y la función miembro estática correspondiente usando la clase Rectangle. El programa también muestra cómo podemos contar las instancias.

Ejercicio: Cuenta bancaria

- Diseñar una clase que representa una cuenta bancaria. Con dos miembros de datos de instancia: el número de cuenta y el saldo. Si bien el usuario de la clase puede inicializar el saldo durante la creación de una instancia, el usuario no puede ingresar el número de cuenta porque debe ser único. Para evitar la duplicación del número de cuenta, utilizar un miembro de datos estáticos, llamado base, que se inicializa a 0 y se incrementa con la apertura de cada nueva cuenta. Agregar 100000 a este miembro de datos estáticos para que sea un número grande.

Ejercicio: Cuenta bancaria

- Debe permitir que la cuenta se cierre cuando finalice el programa (al llamar al destructor, la cuenta se cierra y el saldo restante se envía al banco cliente).
- Debido a la inicialización del número de cuenta debe definir un constructor de parámetros.
- Debe evitar que el sistema cree un constructor copia pues no podemos tener dos cuentas con el mismo número de cuenta (en C++11 declare un constructor de copia y establecerle la palabra clave delete)
- Debe garantizar que no se abra ninguna cuenta con saldo negativo