



UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

► **Curso: Fundamentos de Programación**

► **Docente: Américo Chulluncuy Reynoso**

2025-3

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

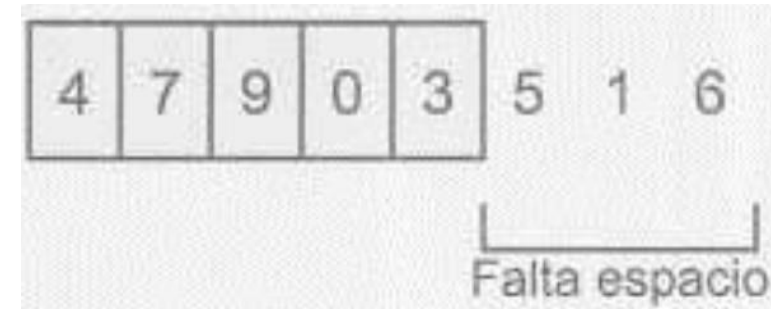
1. Arreglos Estáticos

```
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.

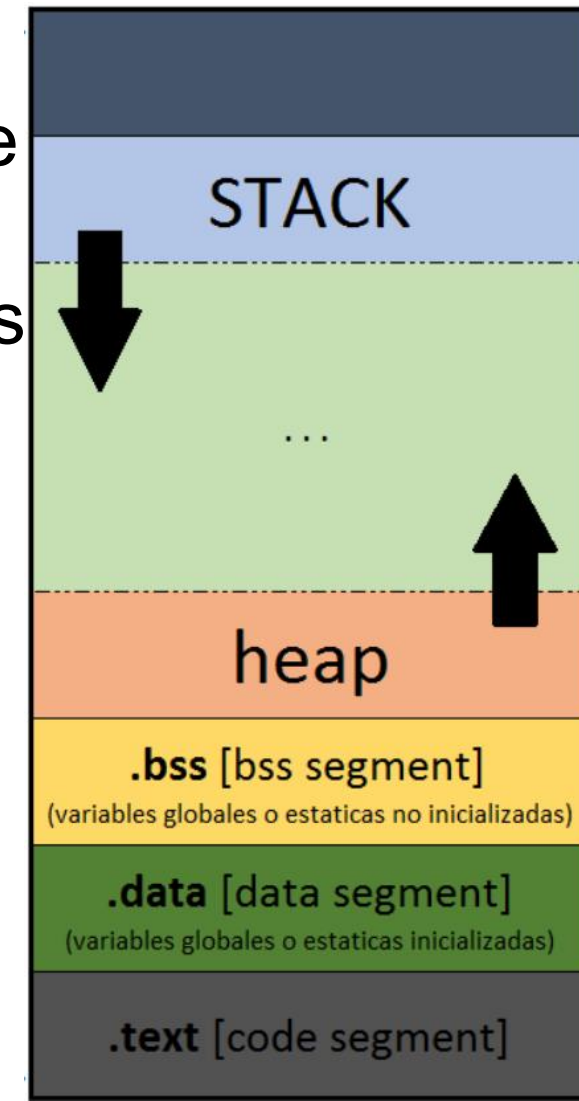
El uso de **memoria dinámica** en **C++** se gestiona de forma explícita:

- 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.

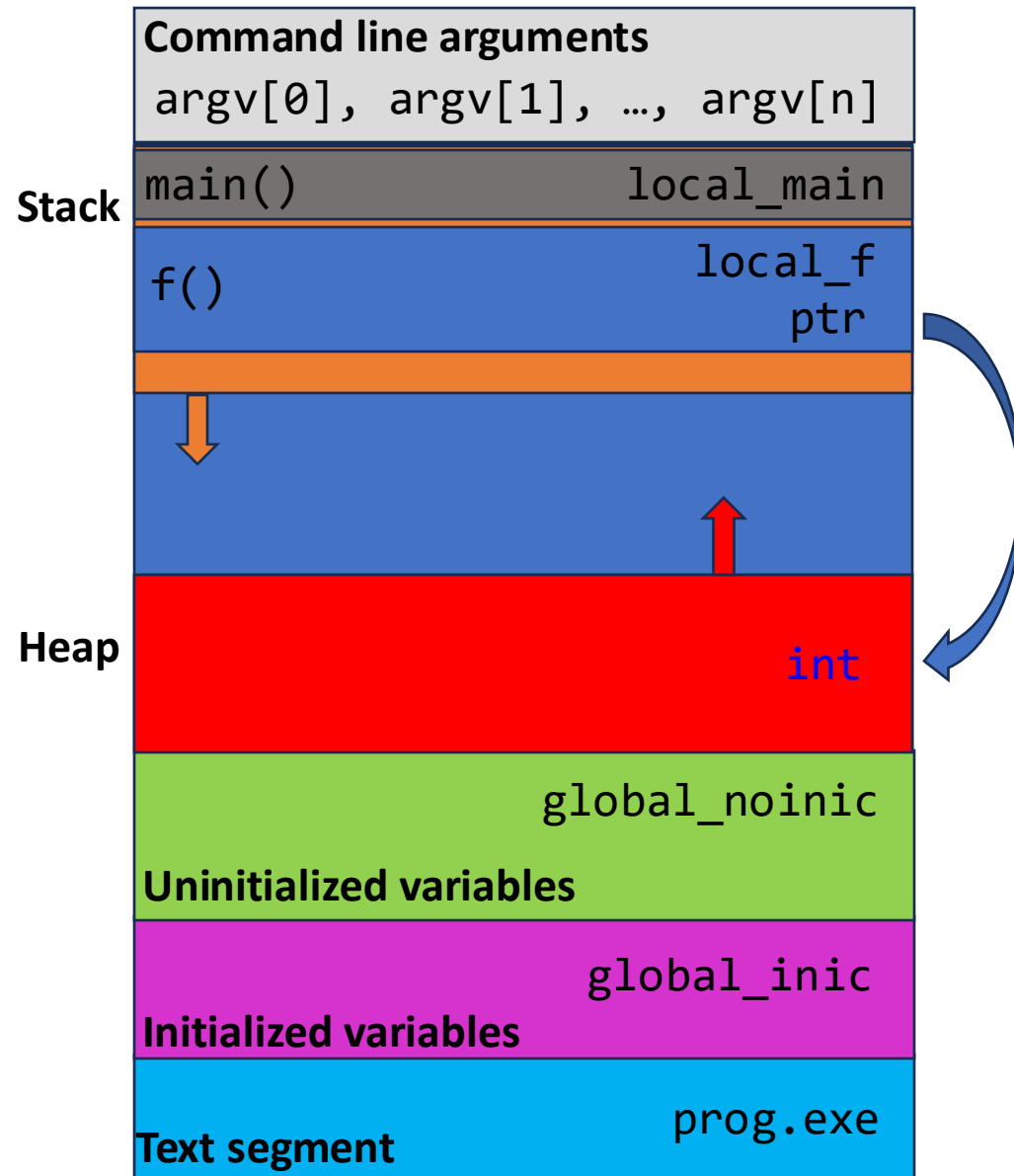
2. Memoria dinámica de la PC

- 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 << "Ingresa 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

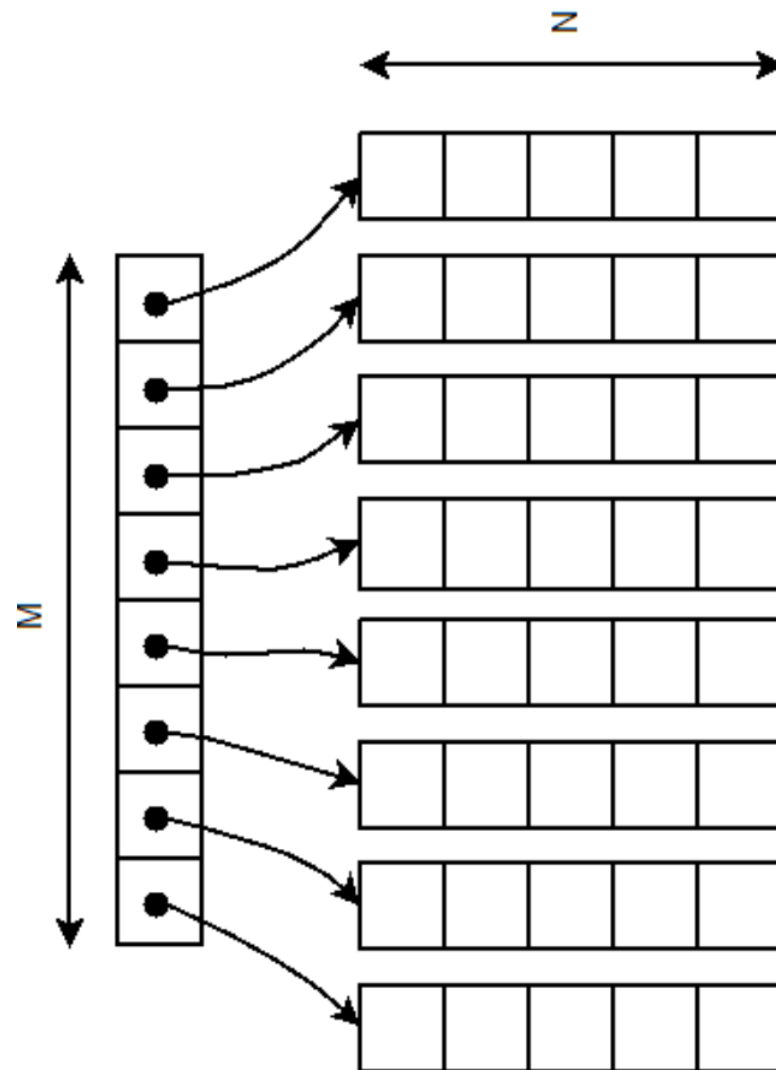
```
#include <iostream>
#include <cstdlib>
using namespace std;

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.