Tema 5. Gestión Dinámica de Memoria

Vicente Benjumea García

Programación-I
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

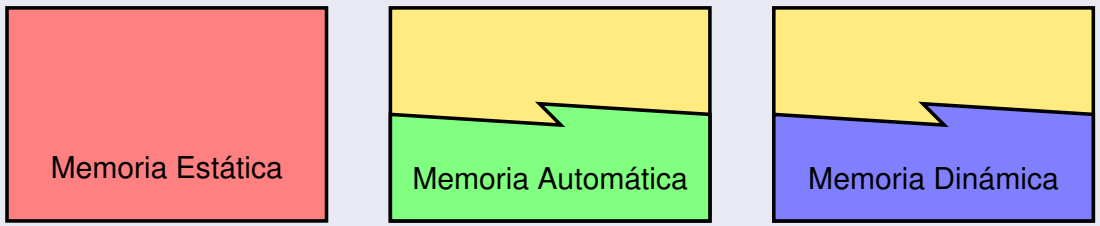
Tema 5. Gestión dinámica de memoria

- Gestión dinámica de memoria.
  - Alojar, realojar y liberar zonas de memoria
  - Listas con número variable de elementos en memoria dinámica.
- Aritmética de punteros.
  - Arrays y aritmética de punteros
  - Buffers de memoria y aritmética de punteros

Gestión dinámica de memoria (I)

Áreas de Memoria durante la ejecución de un programa

- **Memoria estática** (*global*): almacena constantes y datos estáticos y globales, con un tiempo de vida que coincide con el tiempo de ejecución del programa.
- **Memoria automática** (*stack-pila de ejecución*): almacena los parámetros y variables locales automáticas que se crean y destruyen durante la invocación a subprogramas. Gestionada automáticamente por el compilador y el flujo de ejecución del programa.
- **Memoria dinámica** (*heap-montículo*): almacena datos cuyo tiempo de vida está gestionado dinámicamente por el programador, y cuyo acceso se realiza a través de punteros.



Gestión dinámica de memoria (II)

Variables automáticas

- Hasta ahora hemos trabajado con variables automáticas, gestionadas por el compilador.
  - Las variables se **crean** automáticamente cuando el flujo de ejecución entra en el ámbito de su definición.
  - Las variables se **destruyen** automáticamente cuando el flujo de ejecución sale del ámbito donde se declaró la variable.
  - El **tiempo de vida** de las variables automáticas está condicionado por el ámbito de su declaración.
  - El número de variables automáticas y su **capacidad de almacenamiento** está predeterminado por la especificación del programa.

Gestión dinámica de memoria (III)

Variables en memoria dinámica

- Las variables en memoria dinámica permiten **adaptar** la capacidad de almacenamiento de los programas a las **necesidades reales** que surjen durante la ejecución del programa.
- El programador debe gestionar el tiempo de vida de las variables almacenadas en **memoria dinámica**, para ello:
  - El programador debe introducir código para **crear** (alojar, reservar) las variables en memoria dinámica cuando sean necesarias.
  - El programador también debe introducir código para **destruir** (liberar) las variables cuando ya no sean necesarias.
  - El programador debe tener especial cuidado para **no corromper** la gestión ni los datos almacenados en memoria dinámica.
  - El programador debe tener especial cuidado para **no perder** zonas de memoria que haya alojado en la memoria dinámica.
    - Si se pierde la referencia a una zona de memoria, entonces esa zona de memoria no podrá ser utilizada, ni liberada, y por lo tanto se perderá, y se irán agotando los recursos de memoria disponibles.

Gestión dinámica de memoria (IV)

Errores comunes en la gestión de variables de memoria dinámica

- La gestión de las variables en memoria dinámica es bastante **propensa a errores**.

- Es muy fácil equivocarse, y **corromper** la gestión de la memoria dinámica.
  - Es erróneo acceder a datos almacenados en una zona de memoria ya liberada.
  - Es erróneo almacenar datos en una zona de memoria ya liberada.
  - Es erróneo liberar una zona de memoria que ya ha sido liberada anteriormente.

- Es muy fácil equivocarse, y **perder recursos** hasta agotar la memoria disponible.
  - Es erróneo perder la referencia a una zona de memoria reservada que no haya sido liberada.
  - Es erróneo no liberar una zona de memoria que ya no sea necesaria.

Alojar, realojar y liberar zonas de memoria (I)

Gestión dinámica de memoria

- Se debe incluir `#include <stdlib.h>`
- El tipo `void*` representa un puntero general (genérico) a cualquier tipo de datos, y se realiza una conversión (*casting*) automático entre los punteros `void*` y cualquier otro tipo de punteros a otros tipo de datos.
- El operador `sizeof(tipo)` devuelve el tamaño en bytes necesario para almacenar una variable del tipo especificado.
- El **operador de indexación** `[]` se puede utilizar tanto con arrays, como con punteros.

Alojar (reservar) memoria dinámica

- **malloc** reserva (aloja) una zona de memoria del tamaño (en bytes) especificado, y devuelve un puntero a dicha zona de memoria.

```
void* malloc(int tamaño-total-en-bytes);
```

- **calloc** reserva (aloja) una zona de memoria para un array del *número de elementos* especificado, donde cada elemento tiene el *tamaño* (en bytes) especificado, y devuelve un puntero a dicha zona de memoria, inicializada a cero.

```
void* calloc(int num-elementos, int tamaño-del-elemento-en-bytes);
```

Alojar, realojar y liberar zonas de memoria (II)

Realojar memoria dinámica

- **realloc** cambia el tamaño de una zona de memoria previamente reservada (alojada) por *malloc* o *calloc*, al nuevo tamaño (en bytes) especificado, y devuelve un puntero a dicha zona de memoria. En caso de ser necesario realoja una nueva zona de memoria y copia el contenido de la zona anterior.

```
void* realloc(void* ptr, int nuevo-tamaño-total-en-bytes);
```

Liberar memoria dinámica

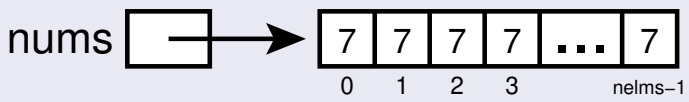
- **free** libera (desaloja) una zona de memoria previamente reservada (alojada) por *malloc*, *calloc* o *realloc*.

```
void free(void* ptr);
```

- En caso de que no haya memoria disponible, estas funciones devuelven `NULL`.
- Para simplificar el aprendizaje, si nuestro programa es correcto y gestiona la memoria adecuadamente, entonces supondremos que siempre tendremos memoria suficiente para resolver el problema especificado.

Alojar, realojar y liberar zonas de memoria. Ejemplo 1

```c
void prueba_1()
{
    int nelms;
    printf("Introduce número de elementos: ");
    scanf(" %d", &nelms);
    //----------------------------
    // int* nums = malloc(nelms * sizeof(int));
    int* nums = calloc(nelms, sizeof(int));
    for (int i = 0; i < nelms; ++i) {
        nums[i] = 7;
    }
    for (int i = 0; i < nelms; ++i) {
        printf(" %d", nums[i]);
    }
    //----------------------------
    free(nums);  // libera el array de números
}
```

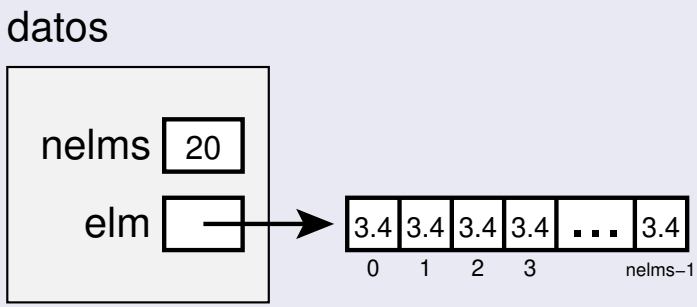


- Se recomienda que el número de elementos y el puntero al array en memoria dinámica sean **agrupados en una misma estructura**.

Alojar, realojar y liberar zonas de memoria. Ejemplo 2

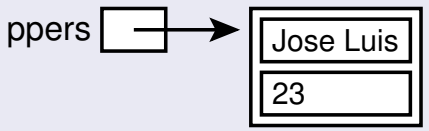
- Se recomienda que el número de elementos y el puntero al array en memoria dinámica sean **agrupados en una misma estructura**.

```
struct Datos {
    int nelms;
    double* elm;
};
void prueba_2()
{
    struct Datos datos;
    printf("Introduce número de elementos: ");
    scanf(" %d", &datos.nelms);
    //----------------------------
    datos.elm = calloc(datos.nelms, sizeof(double));
    for (int i = 0; i < datos.nelms; ++i) {
        datos.elm[i] = 3.4;
    }
    for (int i = 0; i < datos.nelms; ++i) {
        printf(" %lg", datos.elm[i]);
    }
    //----------------------------
    free(datos.elm);  // libera el array de números
    // datos.nelms = 0;
    // datos.elm = NULL;
}
```



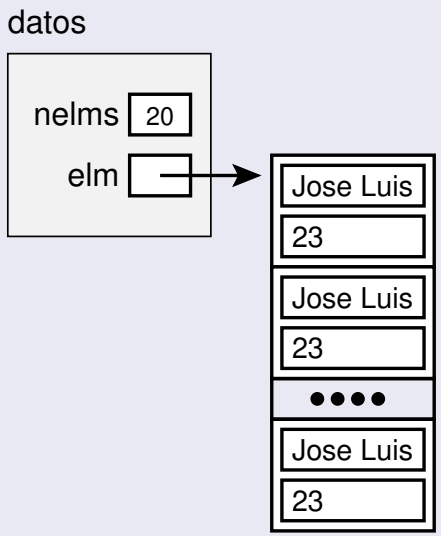
Alojar, realojar y liberar zonas de memoria. Ejemplo 3

```
enum {
    MAXCARS = 31+1,
};
struct Persona {
    char nombre[MAXCARS];
    int edad;
};
void prueba_3()
{
    struct Persona* ppers = malloc(sizeof(struct Persona));
    strncpy(ppers->nombre, "Jose Luis", MAXCARS);
    ppers->nombre[MAXCARS-1] = '\0';
    ppers->edad = 23;
    //----------------------------
    free(ppers);  // libera la persona
}
```



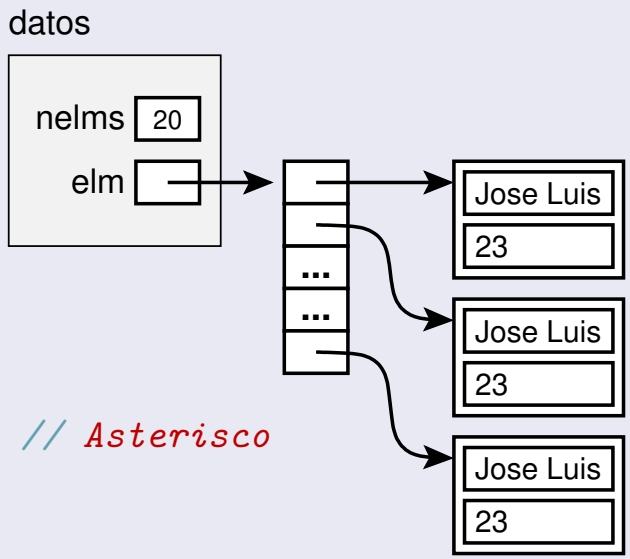
Alojar, realojar y liberar zonas de memoria. Ejemplo 4

```
enum {
    MAXCARS = 31+1,
};
struct Persona {
    char nombre[MAXCARS];
    int edad;
};
struct Datos {
    int nelms;
    struct Persona* elm;
};
void prueba_4()
{
    struct Datos datos;
    printf("Introduce número de personas: ");
    scanf(" %d", &datos.nelms);
    //----------------------------
    datos.elm = calloc(datos.nelms, sizeof(struct Persona));
    for (int i = 0; i < datos.nelms; ++i) {
        strncpy(datos.elm[i].nombre, "Jose Luis", MAXCARS);
        datos.elm[i].nombre[MAXCARS-1] = '\0';
        datos.elm[i].edad = 23;
    }
    //----------------------------
    free(datos.elm);  // libera el array de personas
    // datos.nelms = 0;
    // datos.elm = NULL;
}
```



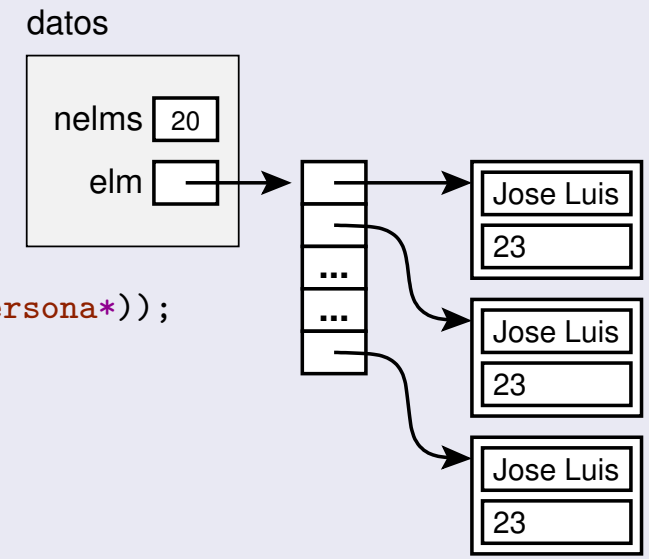
Alojar, realojar y liberar zonas de memoria. Ejemplo 5

```
struct Datos {
    int nelms;
    struct Persona** elm; // Nótese el doble asterisco
};
void prueba_5()
{
    struct Datos datos;
    printf("Introduce número de personas: ");
    scanf(" %d", &datos.nelms);
    //----------------------------
    datos.elm = calloc(datos.nelms, sizeof(struct Persona*)); // Asterisco
    for (int i = 0; i < datos.nelms; ++i) {
        datos.elm[i] = malloc(sizeof(struct Persona));
        strncpy(datos.elm[i]->nombre, "Jose Luis", MAXCARS);
        datos.elm[i]->nombre[MAXCARS-1] = '\0';
        datos.elm[i]->edad = 23;
    }
    //----------------------------
    for (int i = 0; i < datos.nelms; ++i) {
        free(datos.elm[i]);  // libera cada persona
    }
    free(datos.elm);  // libera el array de punteros a personas
    // datos.nelms = 0;
    // datos.elm = NULL;
}
```



Alojar, realojar y liberar zonas de memoria. Ejemplo 6

```
void prueba_6()
{
    struct Datos datos;
    printf("Introduce número de personas: ");
    scanf(" %d", &datos.nelms);
    //----------------------------
    datos.elm = calloc(datos.nelms, sizeof(struct Persona*)); // Asterisco
    for (int i = 0; i < datos.nelms; ++i) {
        datos.elm[i] = malloc(sizeof(struct Persona));
        strncpy(datos.elm[i]->nombre, "Jose Luis", MAXCARS);
        datos.elm[i]->nombre[MAXCARS-1] = '\0';
        datos.elm[i]->edad = 23;
    }
    //----------------------------
    datos.elm = realloc(datos.elm, (2*datos.nelms)*sizeof(struct Persona*));
    for (int i = datos.nelms; i < 2*datos.nelms; ++i) {
        datos.elm[i] = malloc(sizeof(struct Persona));
        strncpy(datos.elm[i]->nombre, "Jose Luis", MAXCARS);
        datos.elm[i]->nombre[MAXCARS-1] = '\0';
        datos.elm[i]->edad = 23;
    }
    datos.nelms *= 2;
    //----------------------------
    for (int i = 0; i < datos.nelms; ++i) {
        free(datos.elm[i]);  // libera cada persona
    }
    free(datos.elm);  // libera el array de punteros a personas
    // datos.nelms = 0;
    // datos.elm = NULL;
}
```



Listas con núm. variable de elementos en m. dinámica (v1)

Programa de notas de alumnos (v5)

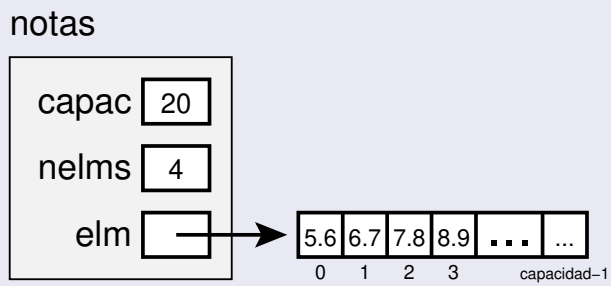
- Lee la **nota** de cada alumno (el número **máximo** de alumnos está limitado por la memoria correspondiente al programa) y muestra si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.
- **El número actual de alumnos será leído de teclado.**

Listas con núm. variable de elementos en m. dinámica (v1)

```
#include <stdio.h>
#include <stdlib.h>

struct LNotas {
    int capacidad;
    int nelms;
    double* elm; // Nótese el asterisco
};

double calc_media(const struct LNotas* v) {
    double media = 0.0;
    if (v->nelms > 0) {
        double suma = 0.0;
        for (int i = 0; i < v->nelms; ++i) {
            suma += v->elm[i];
        }
        media = suma / (double)v->nelms;
    }
    return media;
}
```



Listas con núm. variable de elementos en m. dinámica (v1)

```
void leer_notas(struct LNotas* v)
{
    printf("Introduzca total de alumnos: ");
    scanf(" %d", &v->nelms);
    if (v->nelms <= 0) {
        v->nelms = 0;  // lista vacía
        v->capacidad = 0;
        v->elm = NULL;
        printf("Error\n");
    } else {
        v->capacidad = v->nelms;
        v->elm = calloc(v->capacidad, sizeof(double)); // aloja el array de double
        for (int i = 0; i < v->nelms; ++i) {
            printf("Introduzca la nota del alumno %d: ", i);
            scanf(" %lg", & v->elm[i]);
        }
    }
}
```

Listas con núm. variable de elementos en m. dinámica (v1)

```
void mostrar_notas(const struct LNotas* v, double umbral)
{
    for (int i = 0; i < v->nelms; ++i) {
        if (v->elm[i] >= umbral) {
            printf("Alumno: %d Aprobado\n", i);
        } else {
            printf("Alumno: %d Suspenso\n", i);
        }
    }
}
void destruir_notas(struct LNotas* v)
{
    free(v->elm);                // libera el array
    v->nelms = 0;
    v->capacidad = 0;
    v->elm = NULL;
}
int main()
{
    struct LNotas notas;
    leer_notas(&notas);
    // leer_notas_alternativo(&notas);
    double media = calc_media(&notas);
    mostrar_notas(&notas, media);
    destruir_notas(&notas);
}
```

Listas con núm. variable de elementos en m. dinámica (v1)

```c
void inicializar_notas(struct LNotas* v)
{
    v->nelms = 0;  // lista vacía
    v->capacidad = 16;
    v->elm = calloc(v->capacidad, sizeof(double));  // Aloja el array inicial de double
}
void anyadir_elemento(struct LNotas* v, double n)
{
    if (v->nelms >= v->capacidad) {                          // Si el array está lleno
        v->capacidad *= 2;                                   // Duplica su capacidad
        v->elm = realloc(v->elm, v->capacidad * sizeof(double)); // Duplica capacidad del arr
    }
    v->elm[v->nelms] = n;                          // Añade el elemento al final de la lista
    ++v->nelms;                                    // Incrementa la cuenta de elementos
}
void leer_notas_alternativo(struct LNotas* v)
{
    double valor;
    inicializar_notas(v);
    printf("Introduzca nota del alumno %d (negativo para fin): ", v->nelms);
    scanf(" %lg", &valor);
    while (valor >= 0) {
        anyadir_elemento(v, valor);
        printf("Introduzca nota del alumno %d (negativo para fin): ", v->nelms);
        scanf(" %lg", &valor);
    }
}
```

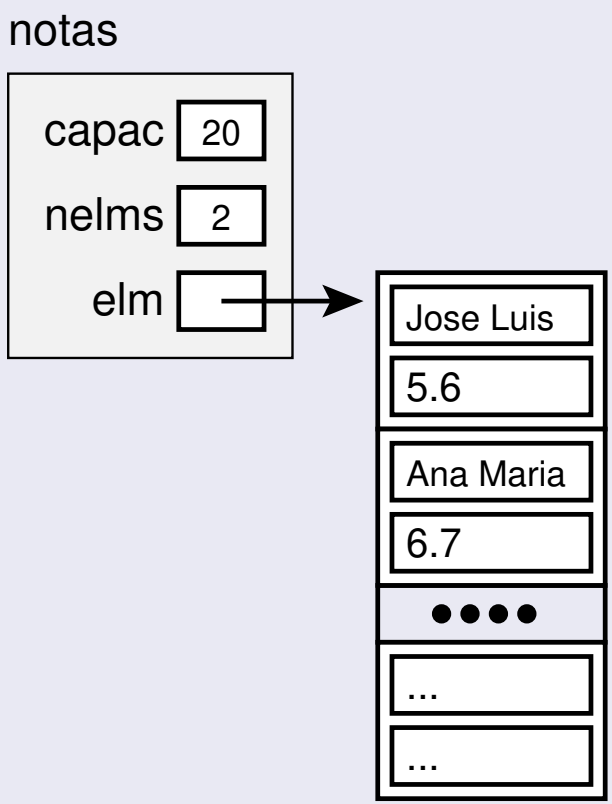
Listas con núm. variable de elementos en m. dinámica (v2)

Programa de notas de alumnos (v6)

- Lee el **nombre y la nota** de cada alumno (el número **máximo** de alumnos está limitado por la memoria correspondiente al programa) y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.
- **El número actual de alumnos será leído de teclado.**
- Versión con array de estructuras Alumno

Listas con núm. variable de elementos en m. dinámica (v2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
enum {
    MAXCARS = 63+1,
};
struct Alumno {
    char nombre[MAXCARS];
    double nota;
};
struct LAlumnos {
    int capacidad;
    int nelms;
    struct Alumno* elm; // Nótese el asterisco
};
double calc_media(const struct LAlumnos* v) {
    double media = 0.0;
    if (v->nelms > 0) {
        double suma = 0.0;
        for (int i = 0; i < v->nelms; ++i) {
            suma += v->elm[i].nota;
        }
        media = suma / (double)v->nelms;
    }
    return media;
}
```



Listas con núm. variable de elementos en m. dinámica (v2)

```
void leer_alumno(struct Alumno* a)
{
    printf("Introduzca el nombre del alumno: ");
    scanf(" %63[^\n]", a->nombre);
    printf("Introduzca la nota del alumno: ");
    scanf(" %lg", &a->nota);
}
void leer_alumnos(struct LAlumnos* v)
{
    printf("Introduzca total de alumnos: ");
    scanf(" %d", &v->nelms);
    if (v->nelms <= 0) {
        v->nelms = 0;  // lista vacía
        v->capacidad = 0;
        v->elm = NULL;
        printf("Error\n");
    } else {
        v->capacidad = v->nelms;
        v->elm = calloc(v->capacidad, sizeof(struct Alumno)); // aloja el array de alumnos
        for (int i = 0; i < v->nelms; ++i) {
            leer_alumno( & v->elm[i] );
        }
    }
}
```

Listas con núm. variable de elementos en m. dinámica (v2)

```
void mostrar_alumno(const struct Alumno* a, double umbral)
{
    if (a->nota >= umbral) {
        printf("Alumno: %s Aprobado\n", a->nombre);
    } else {
        printf("Alumno: %s Suspenso\n", a->nombre);
    }
}
void mostrar_alumnos(const struct LAlumnos* v, double umbral)
{
    for (int i = 0; i < v->nelms; ++i) {
        mostrar_alumno( & v->elm[i], umbral);
    }
}
void destruir_alumnos(struct LAlumnos* v)
{
    free(v->elm);                // libera el array
    v->nelms = 0;
    v->capacidad = 0;
    v->elm = NULL;
}
```

Listas con núm. variable de elementos en m. dinámica (v2)

```
void inicializar_alumnos(struct LAlumnos* v)
{
    v->nelms = 0;  // lista vacía
    v->capacidad = 16;
    v->elm = calloc(v->capacidad, sizeof(struct Alumno));  // Aloja el array inicial de alumnos
}
void anyadir_elemento(struct LAlumnos* v, const struct Alumno* a)
{
    if (v->nelms >= v->capacidad) {                                    // Si el array está lleno
        v->capacidad *= 2;                                             // Duplica su capacidad
        v->elm = realloc(v->elm, v->capacidad * sizeof(struct Alumno)); // Duplica capacidad del array
    }
    v->elm[v->nelms] = *a;                                  // Añade el elemento al final de la lista
    ++v->nelms;                                             // Incrementa la cuenta de elementos
}
```

Listas con núm. variable de elementos en m. dinámica (v2)

```
void leer_alumno_alternativo(struct Alumno* a)
{
    printf("Introduzca el nombre del alumno (fin para terminar): ");
    scanf(" %63[^\n]", a->nombre);
    if (strcmp(a->nombre, "fin") != 0) {
        printf("Introduzca la nota del alumno: ");
        scanf(" %lg", &a->nota);
    }
}
void leer_alumnos_alternativo(struct LAlumnos* v)
{
    inicializar_alumnos(v);
    struct Alumno a;
    leer_alumno_alternativo(&a);
    while (strcmp(a.nombre, "fin") != 0) {
        anyadir_elemento(v, &a);
        leer_alumno_alternativo(&a);
    }
}
int main()
{
    struct LAlumnos alumnos;
    leer_alumnos(&alumnos);
    // leer_alumnos_alternativo(&alumnos);
    double media = calc_media(&alumnos);
    mostrar_alumnos(&alumnos, media);
    destruir_alumnos(&alumnos);
}
```

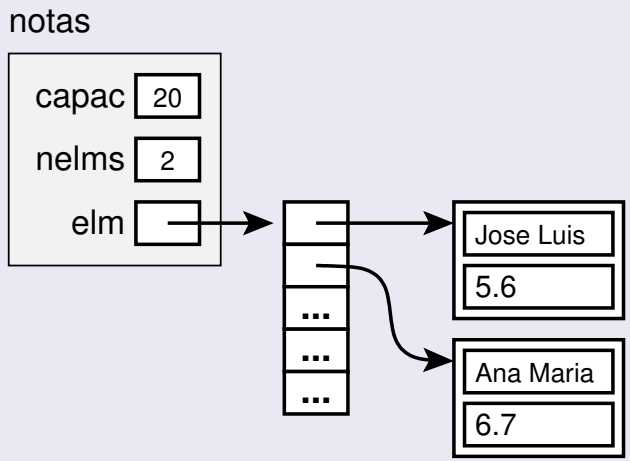
Listas con núm. variable de elementos en m. dinámica (v3)

Programa de notas de alumnos (v7)

- Lee el **nombre y la nota** de cada alumno (el número **máximo** de alumnos está limitado por la memoria correspondiente al programa) y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.
- **El número actual de alumnos será leído de teclado.**
- Versión con array de punteros a estructuras Alumno

Listas con núm. variable de elementos en m. dinámica (v3)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
enum {
    MAXCARS = 63+1,
};
struct Alumno {
    char nombre[MAXCARS];
    double nota;
};
struct LAlumnos {
    int capacidad;
    int nelms;
    struct Alumno** elm; // Nótese el doble asterisco
};
double calc_media(const struct LAlumnos* v) {
    double media = 0.0;
    if (v->nelms > 0) {
        double suma = 0.0;
        for (int i = 0; i < v->nelms; ++i) {
            suma += v->elm[i]->nota;
        }
        media = suma / (double)v->nelms;
    }
    return media;
}
```



Listas con núm. variable de elementos en m. dinámica (v3)

```
void leer_alumno(struct Alumno* a)
{
    printf("Introduzca el nombre del alumno: ");
    scanf(" %63[^\n]", a->nombre);
    printf("Introduzca la nota del alumno: ");
    scanf(" %lg", &a->nota);
}
void leer_alumnos(struct LAlumnos* v)
{
    printf("Introduzca total de alumnos: ");
    scanf(" %d", &v->nelms);
    if (v->nelms <= 0) {
        v->nelms = 0;  // lista vacía
        v->capacidad = 0;
        v->elm = NULL;
        printf("Error\n");
    } else {
        v->capacidad = v->nelms;
        v->elm = calloc(v->capacidad, sizeof(struct Alumno*)); // aloja el array de punteros
        for (int i = 0; i < v->nelms; ++i) {
            v->elm[i] = malloc(sizeof(struct Alumno));         // aloja cada alumno
            leer_alumno( v->elm[i] );
        }
    }
}
```

Listas con núm. variable de elementos en m. dinámica (v3)

```
void mostrar_alumno(const struct Alumno* a, double umbral)
{
    if (a->nota >= umbral) {
        printf("Alumno: %s Aprobado\n", a->nombre);
    } else {
        printf("Alumno: %s Suspenso\n", a->nombre);
    }
}
void mostrar_alumnos(const struct LAlumnos* v, double umbral)
{
    for (int i = 0; i < v->nelms; ++i) {
        mostrar_alumno( v->elm[i], umbral);
    }
}
void destruir_alumnos(struct LAlumnos* v)
{
    for (int i = 0; i < v->nelms; ++i) {
        free(v->elm[i]);          // libera cada alumno
    }
    free(v->elm);                 // libera el array
    v->nelms = 0;
    v->capacidad = 0;
    v->elm = NULL;
}
```

Listas con núm. variable de elementos en m. dinámica (v3)

```
void inicializar_alumnos(struct LAlumnos* v)
{
    v->nelms = 0;  // lista vacía
    v->capacidad = 16;
    v->elm = calloc(v->capacidad, sizeof(struct Alumno*));  // Aloja el array inicial de punteros
}
void anyadir_elemento(struct LAlumnos* v, const struct Alumno* a)
{
    if (v->nelms >= v->capacidad) {                                    // Si el array está lleno
        v->capacidad *= 2;                                             // Duplica su capacidad
        v->elm = realloc(v->elm, v->capacidad * sizeof(struct Alumno*)); // Duplica capacidad del array
    }
    v->elm[v->nelms] = malloc(sizeof(struct Alumno));   // Aloja cada alumno
    *v->elm[v->nelms] = *a;                             // Añade el elemento al final de la lista
    ++v->nelms;                                         // Incrementa la cuenta de elementos
}
```

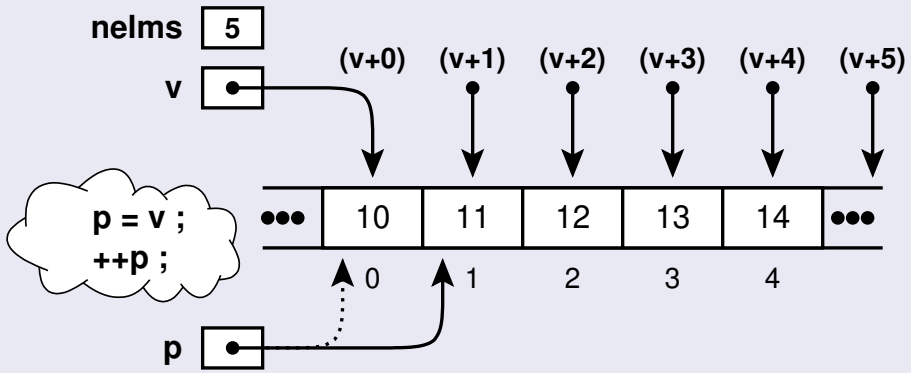
Listas con núm. variable de elementos en m. dinámica (v3)

```
void leer_alumno_alternativo(struct Alumno* a)
{
    printf("Introduzca el nombre del alumno (fin para terminar): ");
    scanf(" %63[^\n]", a->nombre);
    if (strcmp(a->nombre, "fin") != 0) {
        printf("Introduzca la nota del alumno: ");
        scanf(" %lg", &a->nota);
    }
}
void leer_alumnos_alternativo(struct LAlumnos* v)
{
    inicializar_alumnos(v);
    struct Alumno a;
    leer_alumno_alternativo(&a);
    while (strcmp(a.nombre, "fin") != 0) {
        anyadir_elemento(v, &a);
        leer_alumno_alternativo(&a);
    }
}
int main()
{
    struct LAlumnos alumnos;
    leer_alumnos(&alumnos);
    // leer_alumnos_alternativo(&alumnos);
    double media = calc_media(&alumnos);
    mostrar_alumnos(&alumnos, media);
    destruir_alumnos(&alumnos);
}
```

Arrays y aritmética de punteros (NO RECOMENDADA)

- En las operaciones, los arrays **decaen** automáticamente a punteros al primer elemento.
- Incrementar el valor de un puntero hace que referencie al elemento siguiente en el array.
- Decrementar el valor de un puntero hace que referencie al elemento anterior en el array.
- Los punteros se pueden comparar con los op.relac (== , != , > , >= , < , <=).
- El **operador de indexación** [] se puede utilizar tanto con arrays, como con punteros.

```
// Aritmética de Punteros NO RECOMENDADA
// SE RECOMIENDA acceso a través de índices
void leer_vector(int nelms, int v[nelms])
{
    for (int* p = v; p < (v+nelms); ++p) {
      scanf(" %d", p);
    }
}
void mostrar_vector(int nelms, const int v[nelms])
{
    for (const int* p = v; p < (v+nelms); ++p) {
      printf("%d ", *p);
    }
    printf("\n");
}
```




```
for (int i = 0; i < nelms; ++i) {
  printf("%d ", *(v+i));
}
    //  *(v+i) es equivalente a v[i]
```

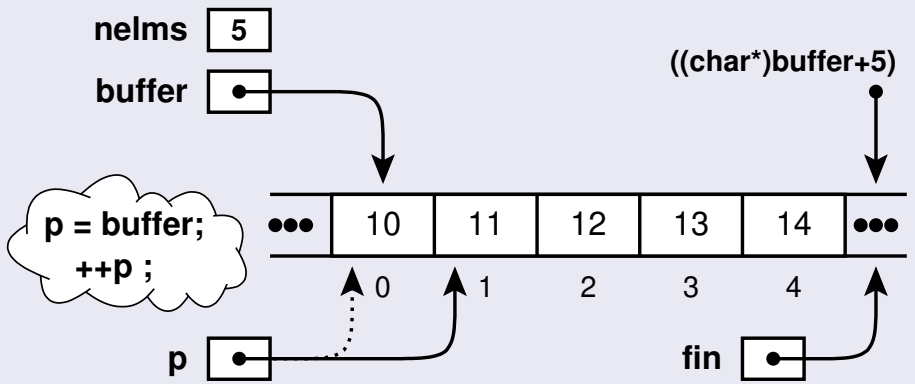
```
void copiar(int nelms, int dst[nelms], const int org[nelms])
{
    const int* o = org; // Equivalente a:  int* o = &org[0];
    for (int *d = dst; (d < (dst+nelms))&&(o < (org+nelms)); ++d) {
        *d = *o;  // Atención al posible error: d = o; // ERROR
        ++o ;
    }
```

```
int main()
{
    int v1[NELMS];
    int v2[NELMS];
    leer_vector(NELMS, v1);
    copiar(NELMS, v2, v1);
    mostrar_vector(NELMS, v2);
```

Buffers y aritmética de punteros (NO RECOMENDADA)

- Se denomina **buffer de memoria** a una zona de memoria contigua, utilizada para almacenar información de tipos diversos, que puede estar alojada en cualquiera de las zonas de memoria del programa. Utilizada habitualmente como puntero (`void*` o `char*`), y suele realizar un tratamiento de bytes (representado con el tipo `char`).
- Ni la aritmética de punteros, ni la indexación, se pueden realizar con `void*`, por lo que se suele hacer una conversión (casting) a `char*` para su tratamiento.

```
// Aritmética de Punteros NO RECOMENDADA
void mostrar_buffer(int nelms, const void* buffer)
{
    const char* fin = (const char*)buffer + nelms;
    for (const char* p = buffer; p < fin; ++p) {
      printf("%#hhx ", *p); // muestra de bytes (char)
    }
    printf("\n");
}
// Aritmética de Punteros NO RECOMENDADA
void copiar_buffer(int nelms, void* dst, const void* org)
{
    char* d = dst;
    const char* fin = (const char*)org + nelms;
    for (const char* p = org; p < fin; ++p) {
        *d = *p; // copia de bytes (char)
        ++d;
    }
}
```




```
int main()
{
  struct Fecha fecha = {1, 2, 2023};
  char v2[sizeof(fecha)];
  copiar_buffer(sizeof(fecha), v2, &fecha);
  copiar_buffer(sizeof(fecha), &fecha, v2);
  mostrar_buffer(sizeof(fecha), &fecha);
  mostrar_buffer(sizeof(fecha), v2);
}
```

Operaciones con Buffers de Memoria

Algunas operaciones proporcionadas por la biblioteca estándar.

- `memset(dest, valor, szbytes)` copia el **valor** (byte) especificado, a la zona de memoria apuntada por **dest**, tantos bytes como indica **sznbytes**.
- `memmove(dest, org, szbytes)` copia de la zona de memoria apuntada por **org**, a la zona de memoria apuntada por **dest**, tantos bytes como indica **szbytes**. Las áreas de memoria sí pueden estar solapadas.
- `memcpy(dest, org, szbytes)` copia de la zona de memoria apuntada por **org**, a la zona de memoria apuntada por **dest**, tantos bytes como indica **szbytes**. Las áreas de memoria **NO** pueden estar solapadas.

```c
#include <string.h>  // se debe incluir <string.h>
enum { NELMS = 10 };
void prueba(int nelms, int array_1[nelms], int array_2[nelms]) {
  memset(array_1, 0, nelms * sizeof(int));         // sizeof(array_1) es INCORRECTO
  memmove(array_2, array_1, nelms * sizeof(int)); // sizeof(array_1) es INCORRECTO
  memcpy(array_2, array_1, nelms * sizeof(int));  // sizeof(array_2) es INCORRECTO
}
int main() {
  int array_1[NELMS];
  int array_2[NELMS];
  memset(array_1, 0, NELMS * sizeof(int));          // copia el valor en byte (no int)
  memmove(array_2, array_1, NELMS * sizeof(int));
  memcpy(array_2, array_1, NELMS * sizeof(int));
}
```