Tema 4. Tipos de Datos Avanzados

Vicente Benjumea García

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

Tema 4. Tipos de datos avanzados

- Tipos de datos.
- Estructuras (registros).
  - Definición de estructuras.
  - Paso de parámetros de estructuras.
  - Operaciones con estructuras.
- Arrays.
  - Definición de arrays.
  - Paso de parámetros de arrays.
  - Operaciones con arrays.
  - Listas con cantidad variable de elementos. Operaciones básicas.
  - Cadenas de caracteres.
- Resolución de problemas usando tipos compuestos.

Esta obra se encuentra bajo una licencia Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) de Creative Commons.

Tipos de Datos

- El **tipo de datos** define las características de los objetos: conjunto de valores que pueden tomar, operaciones que se pueden aplicar, espacio de almacenamiento, etc.
- Los **tipos de datos simples** definen conjuntos de valores formados por elementos **indivisibles** y **ordenados**.
  - Predefinidos: **bool**, **char**, **unsigned**, **int**, **double**, *etc*.
  - Definidos por el programador:
    - **enum** (tipo enumerado): define un nuevo tipo donde el conjunto de valores se especifica como una secuencia de símbolos (enumeración).
- Los **tipos de datos compuestos** (estructurados) definen conjuntos de valores formados por elementos **compuestos** que se pueden manipular como un **todo** o a partir de sus **componentes individuales**.
  - Definidos por el programador:
    - **struct** (estructura o registro): define un nuevo tipo como una **agrupación** de varios componentes que pueden ser de **tipos distintos**, adecuado para procesamiento **individualizado**.
    - **array**: define un nuevo tipo como una **colección** de múltiples elementos del **mismo tipo**, con acceso **parametrizado**, adecuado para procesamiento **iterativo**.
    - Las **cadenas de caracteres** representan *secuencias de caracteres*, con acceso **parametrizado** a los elementos de tipo **char**, y se almacenan como arrays de caracteres.

Estructuras (Registros)

El Tipo Estructura (Registro)

- Es un **tipo definido por el programador**. El programador puede definir tantos tipos nuevos como necesite. También denominado **Registro**.
- El tipo estructura permite definir un **nuevo tipo** como una **agrupación** de varios **componentes** que pueden ser de **distintos tipos**.
- Los componentes se denominan **campos**, y su ámbito de visibilidad se restringe a la propia estructura definida.
- El programador debe especificar el nombre del nuevo tipo, así como el nombre y el tipo de los componentes.
- Se puede **acceder** a cada componente individual de la agrupación mediante su **identificador**. Adecuado para procesamiento **individualizado**.

```
// Definición de Estructura
struct NombreDelNuevoTipoEstructura {
    Tipo1 campo_1;
    Tipo2 campo_1;
    ...
    TipoN campo_n;
} ; // Punto-y-coma
```

```
// Definición de Estructura y Alias (typedef)
typedef struct NombreDelNuevoTipoEstructura {
    Tipo1 campo_1;
    Tipo2 campo_1;
    ...
    TipoN campo_n;
} NombreDelNuevoTipoEstructura ; // Punto-y-coma
```

Definición de Estructuras (Registros)

- Podemos definir un **nuevo tipo** que represente el concepto de *Fecha* como agrupación de *dia*, *mes* y *año*.

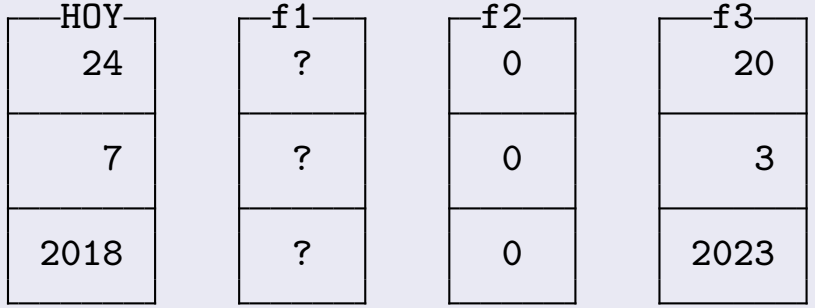
```
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
```

```
typedef struct Fecha {
    int dia;
    int mes;
    int anyo;
} Fecha;
```

```
// Se pueden agrupar los campos
// si son del mismo tipo
struct Fecha {
    int dia, mes, anyo;
};
```

- Los valores del tipo **`struct Fecha`** se componen de tres elementos concretos (de tipo **`int`** cada uno, aunque pueden ser de tipos diferentes) .
- Los identificadores **`dia`**, **`mes`** y **`anyo`** representan los nombres de sus elementos componentes, denominados **campos**, y su ámbito de visibilidad se restringe a la propia estructura definida.
- Podemos declarar constantes, variables y parámetros de dicho tipo
  - Las *llaves-simples* inicializan valores y las *llaves-vacías* inicializan a cero.

```
const struct Fecha HOY = { 24, 7, 2018 };
int main()
{
   struct Fecha f1;        // { ?, ?, ? }
   struct Fecha f2 = {};   // { 0, 0, 0 }
   struct Fecha f3 = {20, 3, 2023};
}
```



Definición de Estructuras (Registros)

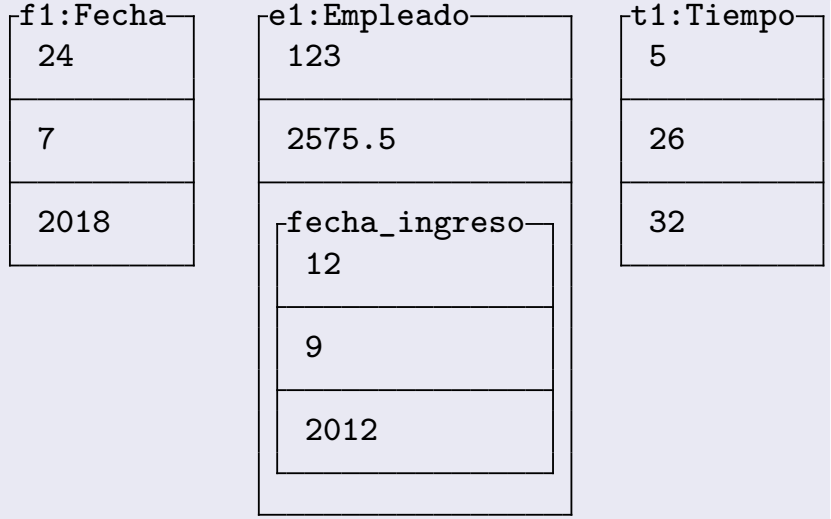
- Los campos de una estructura pueden ser de cualquier tipo de datos, simple o compuesto.
  - Por ejemplo, el campo `fecha_ingreso` del tipo `Empleado` es de tipo `Fecha`.

```
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
```

```
struct Empleado {
    int    codigo;
    double sueldo;
    struct Fecha fecha_ingreso;
};
```

```
struct Tiempo {
    int hor;
    int min;
    int seg;
};
```

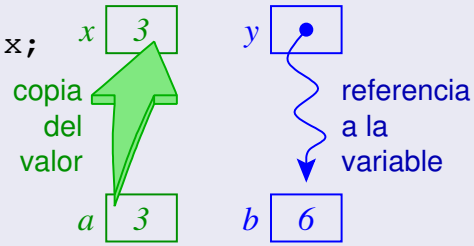
```
int main()
{
  struct Fecha f1 = { 24, 7, 2018 };
  struct Empleado e1 = { 123, 2575.5, { 12, 9, 2012 } };
  struct Tiempo t1 = { 5, 26, 32 };
}
```



Paso de Parámetros de Estructuras (Registros)

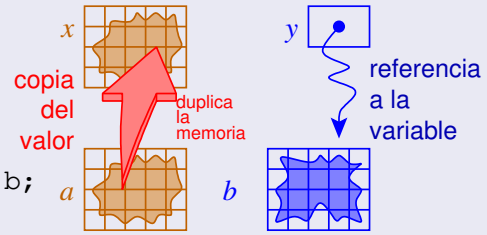
El **paso por valor** de tipos **simples** permite realizar la transferencia de información de *entrada* de forma **eficiente**.

```
void subprograma(int x, int* y)
{
    *y = 2 * x;
}

int main()
{
    int a, b;
    a = 3;
    subprograma(a, &b);
}
```



El **paso por valor** de **estructuras** conlleva una **alta sobrecarga**, ya que se debe duplicar la memoria y copiar el contenido (de gran tamaño).

```
void proc(struct Datos x, struct Datos* y)
{
   // ...
}

int main()
{
   struct Datos a, b;
   // ...
   proc(a, &b);
}
```



Todos los **parámetros de entrada** de **tipos simples** se realizarán mediante el **paso por valor**.

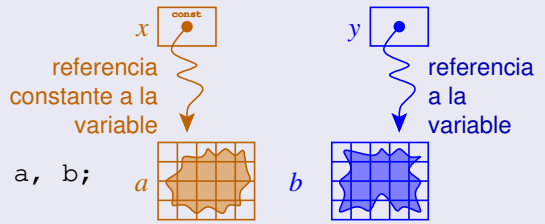
Todos los **parámetros de entrada** de **estructuras** se realizarán mediante el **paso de punteros constante**.

> *El **paso por valor** de **estructuras** **no se debe utilizar**, ya que tiene una **alta sobrecarga**.*

Todos los **parámetros de salida o entrada/salida** se realizarán mediante el **paso de punteros**.

El **paso de punteros constante** de **estructuras** permite realizar la transferencia de información de *entrada* de forma **eficiente**.

```
void proc(const struct Datos* x, struct Datos* y)
{
   // ...
}

int main()
{
   struct Datos a, b;
   // ...
   proc(&a, &b);
}
```



Paso de Parámetros de Estructuras (Registros)

- Todos los **parámetros de salida o entrada/salida** de **estructuras** se realizarán mediante el **paso de punteros**.
- Todos los **parámetros de entrada** de estructuras se realizarán mediante el **paso de punteros constante**.
  - **No se debe utilizar** el **paso por valor** de estructuras, ya que tiene una **alta sobrecarga**, debido a que debe duplicar la memoria y copiar el contenido.

```
void leer_fecha(struct Fecha* f)            // ⇑ Paso de puntero
{
    printf("Introduce dia mes y año: ");
    scanf(" %d %d %d", &f->dia, &f->mes, &f->anyo);
}
void mostrar_fecha(const struct Fecha* f)   // ⇓ Paso de puntero constante
{
    printf("%02d/%02d/%04d\n", f->dia, f->mes, f->anyo);
}
int main()
{
    struct Fecha f1;
    leer_fecha(&f1);
    mostrar_fecha(&f1);
}
```

Operaciones con Estructuras (Registros)

- Un objeto de tipo estructura puede tratarse como un **todo**, o acceder a cada uno de sus **componentes** (campos).
- Un determinado componente podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo.
- Para **acceder** a un determinado componente se **nombra** el objeto seguido por el identificador del **campo** correspondiente, ambos separados por **punto** (**'.'**) (del tipo del componente especificado).
- Si la variable es un **puntero a una estructura**, para **acceder** a un determinado componente se **nombra** la variable seguido por el identificador del **campo** correspondiente, ambos separados por **flecha** (**'->'**) (del tipo del componente especificado).

```c
void mostrar_fecha(const struct Fecha* f)
{
  printf("%02d/%02d/%04d\n", f->dia, f->mes, f->anyo);
  printf("%02d/%02d/%04d\n", (*f).dia, (*f).mes, (*f).anyo);
}
int main()
{
  struct Fecha f = { 24, 7, 2018 };
  mostrar(&f);
}
```

```c
int main()
{
  struct Empleado pepe;
  pepe.codigo = 123;
  pepe.sueldo = 1234.56;
  pepe.fecha_ingreso.dia = 27;
  pepe.fecha_ingreso.mes = 8;
  pepe.fecha_ingreso.anyo = 2018;
}
```

Operaciones con Estructuras (Registros)

Operaciones entre estructuras completas.

- Asignación de *estructuras* completas (=) del mismo tipo.
- Paso como parámetro a subprogramas (⇑⇓*puntero* y ⇓*puntero constante*).
- Devolución por una función. Esta operación es válida, pero no es recomendable.
  - Mejor mediante un parámetro de salida (⇑) con paso de punteros.
- Cualquier otra operación debe ser definida por el programador (programada componente a componente).
  - **No** son válidos los op.relac. (`==` , `!=` , `>` , `>=` , `<` , `<=`), ni `scanf()`, `printf()`.

```c
// Paso de puntero                    ⇑
void leer_fecha(struct Fecha* f)
{
    printf("Introduce dia mes y año: ");
    scanf(" %d %d %d", &f->dia, &f->mes, &f->anyo);
}
// Paso de puntero constante          ⇓
void mostrar_fecha(const struct Fecha* f)
{
    printf("%02d/%02d/%04d\n", f->dia, f->mes, f->anyo);
}
void copiar_fecha(struct Fecha* d, const struct Fecha* o)
{
    *d = *o;  // Asignación de estructuras completas // Atención al posible error: d = o; // ERROR
}
```

```c
int main()
{
    struct Fecha f1, f2, f3;
    leer_fecha(&f1);
    f2 = f1;  // Asignación de estructuras comple
    copiar_fecha(&f2, &f1);
    mostrar_fecha(&f2);
    f3 = (struct Fecha){20, 07, 2023};
    mostrar_fecha(&f3);
}
```

Estructuras (Registros). Ejemplo 1

- Desarrolle un programa que permita introducir los datos de un empleado (código, sueldo y fecha de ingreso) y muestre en pantalla los datos del empleado leídos anteriormente.

Estructuras (Registros). Ejemplo 1 (v1)

Versión-1 (con menos abstracción)

```c
#include <stdio.h>

struct Fecha {
    int dia;
    int mes;
    int anyo;
};
struct Empleado {
    int codigo;
    double sueldo;
    struct Fecha fecha_ingr;
};
```

```c
void leer_empleado(struct Empleado* e)
{
    printf("Introduce código: ");
    scanf(" %d", &e->codigo);
    printf("Introduce sueldo: ");
    scanf(" %lg", &e->sueldo);

    printf("Introduce fecha de ingreso: ");
    printf("Introduce dia mes y año: ");
    scanf(" %d", &e->fecha_ingr.dia);
    scanf(" %d", &e->fecha_ingr.mes);
    scanf(" %d", &e->fecha_ingr.anyo);
}
```

```c
void mostrar_empleado(const struct Empleado* e)
{
    printf("%d ", e->codigo);
    printf("%lg ", e->sueldo);

    printf("%02d/%02d/%04d", e->fecha_ingr.dia,
                             e->fecha_ingr.mes,
                             e->fecha_ingr.anyo);
    printf("\n");
}
```

```c
int main()
{
    struct Empleado e1;
    leer_empleado(&e1);
    mostrar_empleado(&e1);
}
```

Estructuras (Registros). Ejemplo 1 (v2)

Versión-2 (con más abstracción)

```c
#include <stdio.h>

struct Fecha {
    int dia;
    int mes;
    int anyo;
};
struct Empleado {
    int codigo;
    double sueldo;
    struct Fecha fecha_ingr;
};
void mostrar_fecha(const struct Fecha* f)
{
    printf("%02d/%02d/%04d",
           f->dia, f->mes, f->anyo);
}
void mostrar_empleado(const struct Empleado* e)
{
    printf("%d ", e->codigo);
    printf("%lg ", e->sueldo);
    mostrar_fecha( & e->fecha_ingr );
    printf("\n");
}
```

```c
void leer_fecha(struct Fecha* f)
{
   printf("Introduce dia mes y año: ");
   scanf(" %d", &f->dia);
   scanf(" %d", &f->mes);
   scanf(" %d", &f->anyo);
}

void leer_empleado(struct Empleado* e)
{
   printf("Introduce código: ");
   scanf(" %d", &e->codigo);
   printf("Introduce sueldo: ");
   scanf(" %lg", &e->sueldo);
   printf("Introduce fecha de ingreso: ");
   leer_fecha( & e->fecha_ingr );
}

int main()
{
   struct Empleado e1;
   leer_empleado(&e1);
   mostrar_empleado(&e1);
}
```

Estructuras (Registros). Ejemplo 2

- Desarrolle un programa que calcule y muestre la diferencia de tiempo entre dos instantes de tiempo concretos, leídos de teclado, especificados de forma desglosada en *horas*, *minutos* y *segundos*.

Estructuras (Registros). Ejemplo 2

```c
#include <stdio.h>

const int SEGMIN = 60;
const int MINHOR = 60;
const int MAXHOR = 24;
const int SEGHOR = SEGMIN
                   * MINHOR;
struct Tiempo {
    int horas;
    int minutos;
    int segundos;
};
```

```c
int leer_intervalo(int inf, int sup)
{
  int num;
  printf("Introduce valor [%d, %d)", inf, sup);
  scanf(" %d", &num);
  while ( ! ((inf <= num) && (num < sup))) {
    printf("Error. Introduce valor [%d, %d)",
           inf, sup);
    scanf(" %d", &num);
  }
  return num;
}
void leer_tiempo(struct Tiempo* t)
{
    printf("Introduce horas: ");
    t->horas = leer_intervalo(0, MAXHOR);
    printf("Introduce minutos: ");
    t->minutos = leer_intervalo(0, MINHOR);
    printf("Introduce segundos: ");
    t->segundos = leer_intervalo(0, SEGMIN);
}
```

Estructuras (Registros). Ejemplo 2

```
void escribir_tiempo(const struct Tiempo* t)
{
    printf("%02d/%02d/%02d", t->horas, t->minutos, t->segundos);
}
int tiempo_a_seg(const struct Tiempo* t)
{
    return (t->horas * SEGHOR) + (t->minutos * SEGMIN) + (t->segundos);
}
void seg_a_tiempo(int sg, struct Tiempo* t)
{
    t->horas = sg / SEGHOR;
    t->minutos = (sg % SEGHOR) / SEGMIN;
    t->segundos = (sg % SEGHOR) % SEGMIN;
}
void diferencia(const struct Tiempo* t1, const struct Tiempo* t2, struct Tiempo* dif)
{
    seg_a_tiempo(tiempo_a_seg(t2) - tiempo_a_seg(t1), dif);
}
int main()
{
    Tiempo t1, t2, dif;
    leer_tiempo(&t1);
    leer_tiempo(&t2);
    diferencia(&t1, &t2, &dif);
    escribir_tiempo(&dif);
    printf("\n");
}
```

Estructuras (Registros). Ejemplo 3

- Desarrolle un programa que permita introducir los datos de dos empleados (código, sueldo y fecha de ingreso) y muestre los datos del empleado con mayor antigüedad.

Estructuras (Registros). Ejemplo 3

```c
#include <stdio.h>

struct Fecha {
    int dia, mes, anyo;
};
struct Empleado {
    int codigo;
    double sueldo;
    struct Fecha fecha_ingr;
};

void leer_fecha(struct Fecha* f)
{
  printf("Introduce dia mes y año: ");
  scanf(" %d %d %d",
        &f->dia, &f->mes, &f->anyo);
}
```

```c
void leer_empleado(struct Empleado* e)
{
    printf("Introduce código: ");
    scanf(" %d", &e->codigo);
    printf("Introduce sueldo: ");
    scanf(" %lg", &e->sueldo);
    printf("Introduce fecha de ingreso: ");
    leer_fecha( & e->fecha_ingr );
}
void mostrar_fecha(const struct Fecha* f)
{
    printf("%02d/%02d/%04d",
           f->dia, f->mes, f->anyo);
}
void mostrar_empleado(const struct Empleado* e)
{
    printf("%d %lg ", e->codigo, e->sueldo);
    mostrar_fecha( & e->fecha_ingr );
    printf("\n");
}
```

Estructuras (Registros). Ejemplo 3

```
bool es_menor(const struct Fecha* f1,
              const struct Fecha* f2)
{
    bool ok;
    if (f1->anyo < f2->anyo) {
        ok = true;
     } else if (f1->anyo > f2->anyo) {
        ok = false;
     } else if (f1->mes < f2->mes) {
        ok = true;
     } else if (f1->mes > f2->mes) {
        ok = false;
     } else if (f1->dia < f2->dia) {
        ok = true;
     } else if (f1->dia > f2->dia) {
        ok = false;
     } else {
        // fechas iguales
        ok = false;
    }
    return ok;
}
bool mas_antiguo(const struct Empleado* e1, const struct Empleado* e2)
{
    return es_menor( & e1->fecha_ingr, & e2->fecha_ingr);
}
```

```
int main()
{
    struct Empleado e1, e2;
    leer_empleado(&e1);
    leer_empleado(&e2);
    if (mas_antiguo(&e1, &e2)) {
        mostrar_empleado(&e1);
    } else {
        mostrar_empleado(&e2);
    }
}
```

Arrays

El Tipo Array

- Es un **tipo definido por el programador**. El programador puede definir tantos tipos nuevos como necesite.
- El tipo array permite definir un **nuevo tipo** como una **colección** (agregación) de una cantidad **fija** de **elementos** del **mismo tipo**.
- La **cantidad de elementos** de la colección debe ser **constante**, definida en **tiempo de compilación**.
- El programador debe especificar el nombre del nuevo tipo, así como el tipo de los elementos y la cantidad de elementos.
- Se puede **acceder** a cada elemento de la colección de forma **parametrizada**. Adecuado para procesamiento **iterativo**.

- Se puede especificar el tipo **array** en la propia definición de las constantes, variables y parámetros, especificando la cantidad de elementos entre corchetes:

```
// tipo-base  variable  cantidad-de-elementos
        int   practicas[NUMERO_ELEMENTOS] ; // variable de tipo ARRAY de INT
```

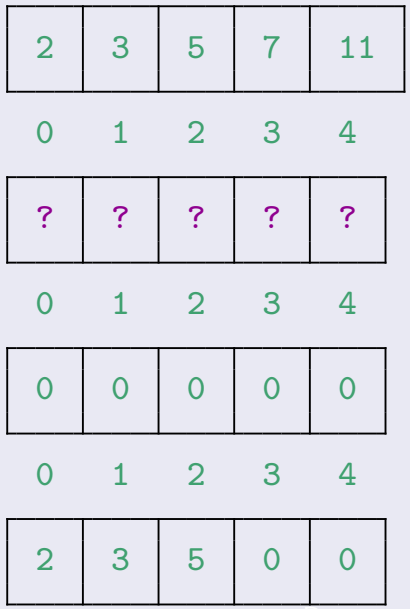
- **typedef** también permite definir un *alias (nombre)* asociado al tipo **array**:

```
// tipo-base  alias  cantidad-de-elementos
typedef int  Numeros[NUMERO_ELEMENTOS] ;
Numeros practicas; // variable de tipo Numeros (ARRAY de INT)
```

Arrays

- En la definición de un tipo Array interviene:
  - El **tipo base** es el tipo de los elementos que constituyen el array. Puede ser cualquier tipo de datos, simple o compuesto.
  - La **cantidad** de elementos que forman la colección. Debe ser **constante**.
- Podemos definir constantes, variables y parámetros de tipo array.
  - Las *llaves* inicializan valores y las *llaves-vacías* inicializan a cero.
- Por ejemplo, podemos definir constantes y variables de tipo array que representen una colección de 5 números enteros.

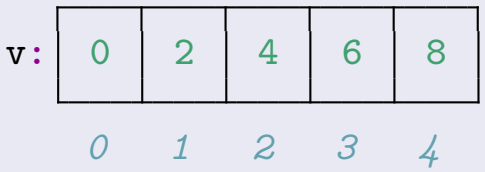
```
enum {
    NELMS = 5, // se debe definir el tamaño del array con un NOMBRE en enum
} ;

const int PRIMOS[NELMS] = { 2, 3, 5, 7, 11 };

int main()
{
   int v1[NELMS];


   int v2[NELMS] = {};


   int v3[NELMS] = { 2, 3, 5 };
```



Arrays. Acceso a los Componentes

- Un objeto de tipo array puede tratarse como un **todo**, o acceder a cada uno de sus **componentes** (elementos).
- Un determinado elemento podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo base.
- **Acceso al iésimo** elemento del array `v` (del tipo base del array):
  - `v[i]` donde `i` ∈ { `0...NELMS-1` }

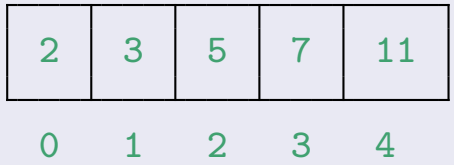
```
enum {
    NELMS = 5,
} ;
int main()
{
    int v[NELMS];
    for (int i = 0; i < NELMS; ++i) {
        v[i] = 2 * i;
    }
    int primer_elemento = v[0];          // 0
    int ultimo_elemento = v[NELMS - 1]; // 8
}
```

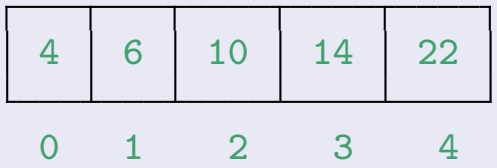


El compilador no comprueba que el valor del índice se encuentre dentro del intervalo válido ⇒ ERRORES GRAVES ⇒ Desbordamiento de Buffer ⇒ Buffer overflow

Arrays. Ejemplo

```
enum {
    NELMS = 5,
} ;


const int PRIMOS[NELMS] = { 2, 3, 5, 7, 11 };

int main()
{
    int v1[NELMS];
    for (int i = 0; i < NELMS; ++i) {
        v1[i] = PRIMOS[i] * 2;
    }

    int v2[NELMS];
    for (int i = 0; i < NELMS; ++i) {
        scanf(" %d", &v2[i]);
    }
    for (int i = 0; i < NELMS; ++i) {
        printf("%d ", v2[i]);
    }
    printf("\n");
}
```





Arrays. Procesamiento iterativo

- Los programas deben **funcionar adecuadamente** incluso cuando se **modifiquen** los valores de las **constantes** que determinan la cantidad de elementos de los arrays.
  - Los arrays se deben manipular con **procesamientos iterativos** (bucles).

```
enum {
    NELMS = 3, // se debe definir el tamaño del array con un NOMBRE en enum
} ;

void procesamiento_iterativo() // procesamiento iterativo -> BIEN
{
    int v[NELMS];                                   //
    for (int i = 0; i < NELMS; ++i) {               //   v:  1 | 2 | 3
        v[i] = i + 1;                               //
    }                                               //       0   1   2
}
void procesamiento_fijo() // procesamiento fijo -> MAL
{
    int v[NELMS]; // Si cambiase el valor           //
    v[0] = 1;     // de la constante NELMS,         //   v:  1 | 2 | 3
    v[1] = 2;     // entonces el programa           //
    v[2] = 3;     // funcionaría MAL                //       0   1   2
}
```

Paso de Parámetros de Arrays (I)

- En el paso de parámetros, el **array pierde su tamaño** y automáticamente **se convierte en un puntero** al primer elemento del array.
  - La **cantidad de elementos** del array se debe **pasar como parámetro (VLA)**.
  - Todos los **parámetros de salida o entrada/salida** de **arrays** se realizarán mediante el paso de **arrays de longitud variable (VLA)**.
  - Todos los **parámetros de entrada** de **arrays** se realizarán mediante el paso de **arrays de longitud variable (VLA) constante**.

El compilador no comprueba que el tamaño de los arrays sea válido
⇒ ERRORES GRAVES ⇒ Desbordamiento de Buffer ⇒ Buffer overflow

```c
// FORMA RECOMENDADA (VLA como parámetro)
void mostrar_vector(int nelms, const int v[nelms])
{
    for (int i = 0; i < nelms; ++i) {
      printf("%d ", v[i]);
    }
    printf("\n");
}

// VLA: "Variable Length Array"
// Este mecanismo se denomina "parámetro VLA"
// Sólo utilizaremos VLA como parámetros
// No utilizaremos VLA como variables
```

```c
// FORMA RECOMENDADA (VLA como parámetro)
void leer_vector(int nelms, int v[nelms])
{
    for (int i = 0; i < nelms; ++i) {
      scanf(" %d", &v[i]);
    }
}
int main()
{
    int v1[NELMS];
    leer_vector(NELMS, v1);
    mostrar_vector(NELMS, v1);
}
```

Paso de Parámetros de Arrays (II)

- Se recomienda NO realizar el paso de parámetros de arrays como punteros.
- Se recomienda NO realizar el paso de parámetros de arrays sin el tamaño.

```c
// FORMA RECOMENDADA (VLA como parámetro)


void leer_vector(int nelms, int v[nelms])
{
    for (int i = 0; i < nelms; ++i) {
      scanf(" %d", &v[i]);
    }
}



void mostrar_vector(int nelms, const int v[nelms])
{
    for (int i = 0; i < nelms; ++i) {
      printf("%d ", v[i]);
    }
    printf("\n");
}

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

```c
// FORMA NO RECOMENDADA (arrays como punteros)
// FORMA NO RECOMENDADA (arrays sin tamaño)
//void leer_vector(int nelms, int* v)
  void leer_vector(int nelms, int v[])
  {
      for (int i = 0; i < nelms; ++i) {
        scanf(" %d", &v[i]);
      }
  }

//void mostrar_vector(int nelms, const int* v)
  void mostrar_vector(int nelms, const int v[])
  {
      for (int i = 0; i < nelms; ++i) {
        printf("%d ", v[i]);
      }
      printf("\n");
  }

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

Estructuras con componentes de tipo Array.

- Es posible definir estructuras con **componentes** de tipo **array**.
  - Se aplica el paso de parámetros de estructuras.
  - **Sí** es válido asignar estructuras completas con componentes de tipo array.
  - Es adecuada para trabajar con arrays con la **cantidad de elementos prefijada**.
  - La cantidad de elementos del array está vinculada al tipo de la estructura, por lo que se puede utilizar el valor de la constante **NELMS** en los bucles, etc.

```c
#include <stdio.h>
enum {
     NELMS = 5,
};
struct Vector {
    int elm[NELMS];
};
void leer_vector(struct Vector* v)
{
    for (int i = 0; i < NELMS; ++i) {
      scanf(" %d", &v->elm[i]);
    }
}
void mostrar_vector(const struct Vector* v)
{
    for (int i = 0; i < NELMS; ++i) {
      printf("%d ", v->elm[i]);
    }
    printf("\n");
}
```

```c
void copiar(struct Vector* dst, const struct Vector* org)
{
    *dts = *org; // Asignación de estructuras completas
    // Atención al posible error: dts = org; // ERROR
}
int main() {
    struct Vector v1;
    struct Vector v2;
    leer_vector(&v1);
    v2 = v1;      // Asignación de estructuras completas
    copiar(&v2, &v1);
    mostrar_vector(&v2);
}
```

Operaciones con Arrays

Operaciones con Arrays completos

- Paso como parámetro a subprogramas. (⇑⇕*VLA* y ⇓*VLA constante*).
  - El **array pierde su tamaño** y **se convierte en un puntero** automáticamente .
- Cualquier otra operación debe ser definida por el programador (programada elemento a elemento).
  - **No** es válida la asignación (=) de *arrays* completos.
    - **Sí** es válido asignar estructuras completas con arrays internos.
  - **No** es válido que una función devuelva un valor de tipo array.
    - Se debe hacer mediante un parámetro de salida (VLA como parámetro no const).
  - **No** son válidos los op.relac. (`==` , `!=` , `>` , `>=` , `<` , `<=`), ni `scanf()`, `printf()`.

```c
void leer_vector(int nelms, int v[nelms])
{
    for (int i = 0; i < nelms; ++i) {
      scanf(" %d", &v[i]);
    }
}
void mostrar_vector(int nelms, const int v[nelms])
{
    for (int i = 0; i < nelms; ++i) {
      printf("%d ", v[i]);
    }
    printf("\n");
}
```

```c
void cp(int nelms, int dst[nelms], const int org[nelms])
{
    for (int i = 0; i < nelms; ++i) {
        dst[i] = org[i];
    }
}
int main() {
    int v1[NELMS];
    int v2[NELMS];
    leer_vector(NELMS, v1);
    cp(NELMS, v2, v1);
    mostrar_vector(NELMS, v2);
}
```

Arrays

Utilidad de los Arrays

Los arrays son útiles en todas aquellas circunstancias en que necesitamos tener **almacenados una colección de valores** (una cantidad fija predeterminada en tiempo de compilación) a los cuales pretendemos **acceder de forma parametrizada**, normalmente para aplicar un **procesamiento iterativo**.

Ejemplo 1. Programa de notas de alumnos (v1)

Lee la nota de cada alumno (la cantidad de alumnos es 20) y muestra si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual a 5.

Utilidad de los Arrays. Ejemplo 1

```c
#include <stdio.h>

const int NALUMNOS = 20;
const double APROBADO = 5.0;

int main()
{
    for (int i = 0; i < NALUMNOS; ++i) {
        printf("Introduzca nota del alumno %d: ", i);
        double nota;
        scanf(" %lg", &nota);
        if (nota >= APROBADO) {
            printf("Alumno: %d Aprobado\n", i);
        } else {
            printf("Alumno: %d Suspenso\n", i);
        }
    }
}
```

Utilidad de los Arrays. Ejemplo 2

Ejemplo 2. Programa de notas de alumnos (v2)

Lee la nota de cada alumno (la cantidad de alumnos es 20) 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**.

Utilidad de los Arrays. Ejemplo 2

```c
#include <stdio.h>

enum {
     NALUMNOS = 20,
};

void leer_notas(int nelms, double notas[nelms])
{
    for (int i = 0; i < nelms; ++i) {
        printf("Introduzca nota del alumno %d: ", i);
        scanf(" %lg", &notas[i]);
    }
}
double calc_media(int nelms, const double notas[nelms])
{
    double media = 0.0;
    if (nelms > 0) {
        double suma = 0.0;
        for (int i = 0; i < nelms; ++i) {
            suma += notas[i];
        }
        media = suma / (double)nelms;
    }
    return media;
}
```

Utilidad de los Arrays. Ejemplo 2

```
void mostrar_notas(int nelms, const double notas[nelms], double umbral)
{
    for (int i = 0; i < nelms; ++i) {
        if (notas[i] >= umbral) {
            printf("Alumno: %d Aprobado\n", i);
        } else {
            printf("Alumno: %d Suspenso\n", i);
        }
    }
}
int main()
{
    double notas[NALUMNOS];
    leer_notas(NALUMNOS, notas);
    double media = calc_media(NALUMNOS, notas);
    mostrar_notas(NALUMNOS, notas, media);
}
```

Arrays Incompletos: Listas con Cantidad Variable de Elementos

- Hay situaciones en las que debemos almacenar una **lista** de elementos, donde la cantidad de elementos puede **variar** durante la ejecución del programa, pero nunca sobrepasará un determinado **límite máximo**.
- Usualmente, la opción más adecuada para gestionar esta estructura de datos suele ser definir un **tipo estructura** que contenga:
  - La **cantidad de elementos** válidos que **actualmente** contiene la lista.
  - Un **array**, del tamaño adecuado al límite máximo de la lista, que almacene los elementos **consecutivamente** al principio.
  - En ocasiones, la estructura puede tener más datos.
- Tiene diversas denominaciones, hay que entender el concepto, que puede surgir en diferentes contextos y denominaciones, y aplicarlo adecuadamente.

```
enum {
    MAX_ELEMENTOS = 100,
};
struct Lista {
    int nelms;                    // cantidad de elementos almacenados en el array
    TipoBase elm[MAX_ELEMENTOS]; // los elementos se almacenan en el array
};
```

Utilidad de las Listas. Ejemplo 3

Ejemplo 3. Programa de notas de alumnos (v3)

- Lee la nota de cada alumno (la cantidad **máxima** de alumnos es 20) 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**.
- **La cantidad máxima de alumnos es 20, pero la cantidad actual de alumnos será leída de teclado.**

Utilidad de las Listas. Ejemplo 3

```
#include <stdio.h>

enum {
     MAX_ALUMNOS = 20,
};

struct Notas {
    int nelms;
    double elm[MAX_ALUMNOS];
};

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

Utilidad de las Listas. Ejemplo 3

```
void leer_notas(struct Notas* notas)
{
    printf("Introduzca total de alumnos: ");
    scanf(" %d", &notas->nelms);
    if (notas->nelms <= 0 || notas->nelms > MAX_ALUMNOS) {
        notas->nelms = 0;  // lista vacía
        printf("Error\n");
    } else {
        for (int i = 0; i < notas->nelms; ++i) {
            printf("Introduzca nota del alumno %d: ", i);
            scanf(" %lg", &notas->elm[i]);
        }
    }
}
```

Utilidad de las Listas. Ejemplo 3

```
void mostrar_notas(const struct Notas* notas, double umbral)
{
    for (int i = 0; i < notas->nelms; ++i) {
        if (notas->elm[i] >= umbral) {
            printf("Alumno: %d Aprobado\n", i);
        } else {
            printf("Alumno: %d Suspenso\n", i);
        }
    }
}
int main()
{
    struct Notas notas;
    leer_notas(&notas);
    double media = calc_media(&notas);
    mostrar_notas(&notas, media);
}
```

Utilidad de las Listas. Ejemplo 3

```
void anyadir_elemento(struct Notas* notas, double valor)
{
    if (notas->nelms < MAX_ALUMNOS) {       // Si el nuevo elemento cabe en el array
        notas->elm[ notas->nelms ] = valor; // Añade el elemento al final de la lista
        ++notas->nelms;                     // Incrementa la cuenta de elementos
    } else {
        printf("Error\n");
    }
}
void leer_notas_alternativo(struct Notas* notas)
{
    double valor;
    notas->nelms = 0;  // lista vacía
    printf("Introduzca nota del alumno %d (negativo para fin): ", notas->nelms);
    scanf(" %lg", &valor);
    while (valor >= 0) {
        anyadir_elemento(notas, valor);
        printf("Introduzca nota del alumno %d (negativo para fin): ", notas->nelms);
        scanf(" %lg", &valor);
    }
}
```

Búsqueda Lineal o Secuencial (v1)

- Adecuada como mecanismo de **búsqueda general** en colecciones de datos **sin organización** conocida.
- Si encuentra el elemento buscado, entonces devuelve el índice donde se encuentra el elemento en el array, en otro caso devuelve un índice con valor fuera de límites.
  - Procede comparando consecutivamente el elemento a buscar con todos los elementos de la colección, hasta que lo encuentre, o hasta que haya comparado todos los elementos.

```
int buscar(const struct Lista* lista, int x)
{
    int idx = -1;
    for (int i = 0; (i < lista->nelms) && (idx < 0); ++i) {
        if (x == lista->elm[i]) {
            idx = i;
        }
    }
    return idx;
}
```

Búsqueda Lineal o Secuencial (v2 y v3)

Buscar Alternativo (v2)

```
int buscar(const struct Lista* lista, int x)
{
    int idx = -1;
    bool encontrado = false;
    for (int i = 0; (i < lista->nelms) && ( ! encontrado); ++i) {
        if (x == lista->elm[i]) {
            idx = i;
            encontrado = true;
        }
    }
    return idx;
}
```

Buscar Alternativo (v3)

```
int buscar(const struct Lista* lista, int x)
{
    int i = 0;
    // Evaluación en CORTOCIRCUITO
    while ((i < lista->nelms) && (x != lista->elm[i])) {
        ++i;
    }
    if (i == lista->nelms) {
        i = -1;
    }
    return i;
}
```

Operaciones Básicas con Listas (I)

Tipo Lista

```
enum {
     MAX_ELEMENTOS = 100,
};
struct Lista {
    int nelms;
    int elm[MAX_ELEMENTOS];
};
```

Añadir elemento al final

```
void anyadir(struct Lista* lista, int valor, bool* ok)
{
    if (lista->nelms < MAX_ELEMENTOS) {
        lista->elm[ lista->nelms ] = valor;
        ++lista->nelms;
        *ok = true;
    } else {
        *ok = false;
    }
}
```

Operaciones Básicas con Listas (II)

Eliminar elemento desordenando

```
void eliminar(struct Lista* lista, int valor, bool* ok)
{
    int pos = buscar(lista, valor);
    if (0 <= pos && pos < lista->nelms) {
        lista->elm[pos] = lista->elm[lista->nelms-1];
        --lista->nelms;
        *ok = true;
    } else {
        *ok = false;
    }
}
```

Operaciones Básicas con Listas (III)

Añadir elemento ordenado

```
int buscar_posicion(const struct Lista* lista, int valor)
{
    int pos = lista->nelms;
    for (int i = 0; (i < lista->nelms) && (pos == lista->nelms); ++i) {
        if (valor < lista->elm[i]) {
            pos = i;
        }
    }
    return pos;
}
void anyadir_ord(struct Lista* lista, int valor, bool* ok)
{
    if (lista->nelms < MAX_ELEMENTOS) {
        int pos = buscar_posicion(lista, valor);
        for (int i = lista->nelms-1; i >= pos; --i) {
            lista->elm[i+1] = lista->elm[i];
        }
        lista->elm[pos] = valor;
        ++lista->nelms;
        *ok = true;
    } else {
        *ok = false;
    }
}
```

Operaciones Básicas con Listas (IV)

Eliminar elemento manteniendo el orden

```
void eliminar_ord(struct Lista* lista, int valor, bool* ok)
{
    int pos = buscar(lista, valor);
    if (0 <= pos && pos < lista->nelms) {
        for (int i = pos+1; i < lista->nelms; ++i) {
            lista->elm[i-1] = lista->elm[i];
        }
        --lista->nelms;
        *ok = true;
    } else {
        *ok = false;
    }
}
```

Operaciones Básicas con Listas (V)

Eliminar todos los elementos que cumplen condición

```
// Elimina todos los elementos que cumplen una determinada condición.
// En este caso, elimina todos los elementos que son iguales al valor
// recibido como parámetro.

void eliminar_todos(struct Lista* lista, int valor)
{
    int j = 0;  // j contiene la posición donde almacenar los elementos válidos
    for (int i = 0; i < lista->nelms; ++i) {  // para cada elemento original en pos i
        if ( lista->elm[i] != valor ) {       // si el elemento original es válido
            lista->elm[j] = lista->elm[i];    // almacenar elemento original en pos j
            ++j;                              // incrementa pos j para siguiente válido
        }
    }
    lista->nelms = j;  // j contiene la cantidad de elementos válidos guardados
}
```

Operaciones Básicas con Listas (VI)

Ordenación por Burbuja

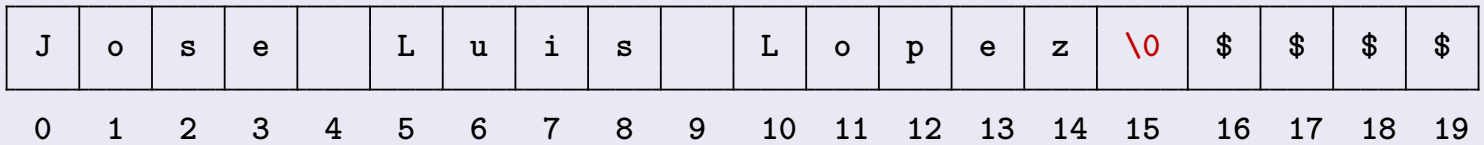
```
// Código simplificado para facilitar su estudio

void ordenar_burbuja(struct Lista* lista)
{
    for (int k = 0; k < lista->nelms-1; ++k) {
        for (int i = 0; i < lista->nelms-1; ++i) {
            if (lista->elm[i] > lista->elm[i+1]) {
                int aux = lista->elm[i];
                lista->elm[i] = lista->elm[i+1];
                lista->elm[i+1] = aux;
            }
        }
    }
}
```

Cadenas de Caracteres (I)

- Las **cadenas de caracteres** representan una sucesión o **secuencia de caracteres**.
- Es un tipo de datos muy versátil, ya que sirve para representar información muy diversa:
  - Representa información **textual** (caracteres).
  - **Entrada** de datos y **salida** de resultados en forma de secuencia de caracteres.
  - Representa información **abstracta** por medio de una secuencia de caracteres.

Cadenas de Caracteres (II)

- Las **cadenas de caracteres** se representan, almacenan y manipulan mediante **arrays** de **char** de tamaño suficiente para almacenar todos los caracteres, según la especificación del problema.
- El caracter **nulo** (`'\0'`) es el caracter **terminador** de la cadena de caracteres, y sirve para indicar el final de la cadena de caracteres dentro del *array de char*.



- El tamaño del array determina la cantidad de caracteres que se pueden almacenar y manipular, considerando el caracter nulo terminador.
  - Siempre debemos utilizar arrays con un elemento adicional a la cantidad máxima de caracteres especificado.
- Una cadena de caracteres **literal** se representa mediante una sucesión de caracteres entre **comillas dobles**: `"esto es un ejemplo"`.
  - La cadena de caracteres vacía se representa así: `""`.

Cadenas de Caracteres (III)

```c
#include <stdio.h>

enum {
    MAXCARS = 31+1, // 31 es el numero máximo de caracteres + 1 caracter nulo terminador
};

const char AUTOR_1[MAXCARS] = "Jose Luis";  // Array constante de 32 caracteres

const char AUTOR_2[MAXCARS] = { 'J', 'o', 's', 'e', ' ', 'L', 'u', 'i', 's', '\0' };

int main()
{
  char nombre_1[MAXCARS] = "Jose Luis Lopez";  // Array con 32 caracteres: Jose Luis Lopez\0
  char nombre_2[MAXCARS] = ""; // cadena vacía // Array con 32 caracteres: cadena vacía

  char nombre_3[MAXCARS] = { 'J', 'o', 's', 'e', ' ', 'L', 'u', 'i', 's', '\0' };
  char nombre_4[MAXCARS] = {}; // cadena vacía // Array con 32 caracteres: cadena vacía

  char nombre_5[MAXCARS];  // sin inicializar  // Array con 32 caracteres: basura
}
```

Paso de Parámetros de Cadenas de Caracteres

- Las cadenas de caracteres se almacenan en **arrays de char**, por lo tanto, el paso de parámetros se realiza como el paso de parámetros de arrays.
  - Se debe pasar el tamaño del array como parámetro.
  - El parámetro se debe pasar como **array de longitud variable (VLA)**, y en el caso de entrada, además, **constante**.

```
enum {
    MAXCARS = 31+1, // maximo 31 caracteres + 1 terminador
};
// FORMA RECOMENDADA (VLA como parámetro)
void copiar(int maxcars, char dst[maxcars], const char org[maxcars])
{
    int i;
    for (i = 0; (i < maxcars-1)&&(org[i] != '\0'); ++i) {
        dst[i] = org[i];
    }
    dst[i] = '\0';
}
```

```
int main()
{
   char texto1[MAXCARS] = "hola, adios";
   char texto2[MAXCARS];
   copiar(MAXCARS, texto2, texto1);
}
```

Operaciones con Cadenas de Caracteres

Salida de cadenas de caracteres

- `printf("%s", texto);` muestra la salida en pantalla. El especificador de formato `"%s"` especifica salida de cadenas de caracteres (*string*).
- `printf("%25s", texto);` el especificador de formato `"%25s"` especifica salida de cadenas de caracteres (*string*), con **anchura de campo 25**.

```c
#include <stdio.h>
enum {
    MAXCARS = 127+1, // maximo 127 caracteres + 1 terminador
};
int main()
{
    char texto1[MAXCARS] = "Jose Luis";

    printf("%s\n", texto1);     // muestra "Jose Luis" en pantalla
    printf("%15s\n", texto1);   // muestra "    Jose Luis" en pantalla
}
```

Operaciones con Cadenas de Caracteres

Entrada de cadenas de caracteres

- `scanf(" %25s", destino);` elimina espacios iniciales, y lee de teclado una cadena de caracteres, hasta leer como máximo **25 caracteres**, o hasta leer **espacio en blanco** o **salto de línea**, y la almacena en el **array de char destino**.
- `scanf(" %25[^\n]", destino);` elimina los espacios iniciales, y lee de teclado una cadena de caracteres, hasta leer como máximo **25 caracteres**, o hasta leer **salto de línea**, y la almacena en el **array de char destino**.
  - El **espacio antes de % es importante** para eliminar los espacios iniciales.

```
char palabra[MAXCARS];
char texto[MAXCARS];

scanf(" %127s", palabra);    // lee una palabra de teclado hasta espacio
scanf(" %127[^\n]", texto);  // lee un texto de teclado hasta salto de línea (\n)

printf("Palabra: [%s]\n", palabra);
printf("Texto: [%s]\n", texto);
```

Operaciones con Cadenas de Caracteres

- La función `strlen(texto)` devuelve la longitud de una cadena de caracteres, sin incluir el caracter nulo terminador.

```
#include <string.h>
enum {
    MAXCARS = 127+1, // maximo 127 caracteres + 1 terminador
};
char texto1[MAXCARS] = "hola y adios";

int longitud = strlen(texto1);      // devuelve 12
int longitud = strlen("Jose Luis"); // devuelve 9
```

- El subprograma `strncpy(destino, origen, maxcars)` permite copiar una cadena de caracteres de origen a destino, como máximo maxcars.
  - Debemos **poner explícitamente** el **terminador** (`'\0'`) en el **último elemento** del array `destino`.

```
#include <string.h>
enum {
    MAXCARS = 127+1, // maximo 127 caracteres + 1 terminador
};
char texto_origen[MAXCARS] = "hola y adios";
char texto_destino[MAXCARS];

strncpy(texto_destino, texto_origen, MAXCARS); // copia de origen a destino
texto_destino[MAXCARS-1] = '\0'; // se debe poner explícitamente el terminador '\0'
```

Operaciones con Cadenas de Caracteres. Comparación Lexicográfica

- La comparación lexicográfica es la comparación de las palabras del diccionario.

```
#include <string.h>
int res = strcmp(texto1, texto2);             // compara hasta terminador
int res = strncmp(texto1, texto2, maxcars);   // compara como máximo maxcars
```

- Comparación lexicográfica sin diferenciar mayúsculas de minúsculas.

```
int res = strcasecmp(texto1, texto2);             // compara hasta terminador
int res = strncasecmp(texto1, texto2, maxcars);   // compara como máximo maxcars
```

- Devuelven tres posibles valores (negativo, cero, positivo):
  - si **texto1 < texto2** entonces **res < 0**
  - si **texto1 == texto2** entonces **res == 0**
  - si **texto1 > texto2** entonces **res > 0**

```
char texto[MAXCARS];
if (strcmp(texto, "fin") < 0) {  /* texto es menor que "fin" */ }
if (strcmp(texto, "fin") == 0) { /* texto es igual que "fin" */ }
if (strcmp(texto, "fin") > 0) {  /* texto es mayor que "fin" */ }

if (strcasecmp(texto, "fin") == 0) { /* TEXTO ES IGUAL QUE "FIN" */ }
if (strncmp(&texto[5], "fin", 3) == 0) { /* texto[5],[6],[7] es igual que "fin" */ }
```

Cadenas de Caracteres. Ejemplos

1. Programa que lee una secuencia de palabras, delimitadas por espacios en blanco y/o saltos de línea, finalizada por la palabra `"fin"`, y muestra en pantalla la cantidad de palabras que comienzan por letra vocal.
2. Programa que lee una palabra, la transforma a mayúsculas, y finalmente la muestra en pantalla.
3. Programa que lee una palabra (formada por letras minúsculas), y escribe su plural según las siguientes reglas:
   - Si acaba en vocal se le añade la letra `'s'`.
   - Si acaba en consonante se le añaden las letras `"es"`. Si la consonante es la letra `'z'`, se sustituye por la letra `'c'`
   - Suponemos que la palabra introducida es correcta.
4. Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es "palíndromo" y falso en caso contrario.

- La palabras de estos ejercicios tendrán un máximo de 31 caracteres.

Cadenas de Caracteres. Ejemplo 1

```c
#include <stdio.h>
#include <string.h>
enum {
    MAXCARS = 31+1, // maximo 31 caracteres + 1 terminador
};
const char FIN[MAXCARS] = "fin";

bool es_vocal(char c)
{
    return (c == 'a') || (c == 'e') || (c == 'i')
        || (c == 'o') || (c == 'u');
}
void procesar(int maxcars, const char palabra[maxcars], int* cnt)
{
    if (es_vocal(palabra[0])) {
        ++ *cnt;
    }
}
int leer_secuencia()
{
    int cnt = 0;
    char palabra[MAXCARS];
    printf("Introduce secuencia de palabras hasta %s:\n", FIN);
    scanf(" %31s", palabra);
    while (strcmp(palabra, FIN) != 0) {
        procesar(MAXCARS, palabra, &cnt);
        scanf(" %31s", palabra);
    }
    return cnt;
}
void mostrar_resultado(int cnt)
{
    printf("Resultado: %d\n", cnt);
}

int main()
{
    int cnt = leer_secuencia();
    mostrar_resultado(cnt);
}
```

Cadenas de Caracteres. Ejemplo 2

```c
#include <stdio.h>
#include <string.h>

enum {
    MAXCARS = 31+1, // maximo 31 caracteres + 1 terminador
};

void leer(int maxcars, char palabra[maxcars])
{
    printf("Introduce palabra: ");
    scanf(" %31s", palabra);
}
void mostrar(int maxcars, const char palabra[maxcars])
{
    printf("Mayusculas: %s\n", palabra);
}
char mayuscula(char letra)
{
    if ((letra >= 'a') && (letra <= 'z')) {
        letra = letra - 'a' + 'A';
    }
    return letra;
}
void mayusculas(int maxcars, char palabra[maxcars])
{
    for (int i = 0; (i < maxcars)&&(palabra[i] != '\0'); ++i) {
        palabra[i] = mayuscula(palabra[i]);
    }
}
```

```c
int main()
{
  char palabra[MAXCARS];
  leer(MAXCARS, palabra);
  mayusculas(MAXCARS, palabra);
  mostrar(MAXCARS, palabra);
}
```

Cadenas de Caracteres. Ejemplo 3

```c
#include <stdio.h>
#include <string.h>

enum {
    MAXCARS = 31+1, // maximo 31 caracteres + 1 terminador
};

bool es_vocal(char c)
{
    return (c == 'a') || (c == 'e') || (c == 'i')
        || (c == 'o') || (c == 'u');
}
void plural(int maxcars, char palabra[maxcars])
{
    int longitud = strlen(palabra);
    if ((longitud > 0)&&(longitud+3 < maxcars)) {
        if (es_vocal(palabra[longitud - 1])) {
            strncpy(&palabra[longitud], "s", maxcars-longitud);
        } else {
            if (palabra[longitud - 1] == 'z') {
                palabra[longitud - 1] = 'c';
            }
            strncpy(&palabra[longitud], "es", maxcars-longitud);
        }
        palabra[maxcars-1] = '\0';
    }
}
```

```c
int main()
{
  char palabra[MAXCARS];
  printf("Introduce  palabra: ");
  scanf(" %31s", palabra);
  plural(palabra);
  printf("Palabra: %s\n", palabra);
}
```

Cadenas de Caracteres. Ejemplo 4

- Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es "palíndromo" y falso en caso contrario.

```
bool es_palindromo(int maxcars, const char palabra[maxcars])
{
    bool ok = false;
    int longitud = strlen(palabra);
    if (longitud > 0) {
        int i = 0;
        int j = longitud - 1;
        while ((i < j) && (palabra[i] == palabra[j])) {
            ++i;
            --j;
        }
        ok = i >= j;
    }
    return ok;
}
```

Resolución de Problemas Usando Tipos Compuestos

Programa de notas de alumnos (v4)

- Lee el **nombre y la nota** de cada alumno (la cantidad **máxima** de alumnos es 20) 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**.
- **La cantidad máxima de alumnos es 20, pero la cantidad actual de alumnos será leída de teclado.**

Resolución de Problemas Usando Tipos Compuestos (I)

```c
#include <stdio.h>
#include <string.h>
enum {
    MAX_ALUMNOS = 20,
    MAXCARS = 63+1,
};
struct Alumno {
    char nombre[MAXCARS];
    double nota;
};
struct LAlumnos {
    int nelms;
    struct Alumno elm[MAX_ALUMNOS];
};
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;
}
```

Resolución de Problemas Usando Tipos Compuestos (II)

```
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 > MAX_ALUMNOS) {
        v->nelms = 0;  // lista vacía
        printf("Error\n");
    } else {
        for (int i = 0; i < v->nelms; ++i) {
            leer_alumno( & v->elm[i]);
        }
    }
}
```

Resolución de Problemas Usando Tipos Compuestos (III)

```
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);
    }
}
int main()
{
    struct LAlumnos alumnos;
    leer_alumnos(&alumnos);
    double media = calc_media(&alumnos);
    mostrar_alumnos(&alumnos, media);
}
```

Resolución de Problemas Usando Tipos Compuestos (IV)

```
void anyadir_elemento(struct LAlumnos* v, const struct Alumno* a)
{
    if (v->nelms < MAX_ALUMNOS) {   // Si el nuevo elemento cabe en el array
        v->elm[v->nelms] = *a;      // Añade el elemento al final de la lista
        ++v->nelms;                 // Incrementa la cuenta de elementos
    } else {
        printf("Error\n");
    }
}
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)
{
    struct Alumno a;
    v->nelms = 0;  // lista vacía
    leer_alumno_alternativo(&a);
    while (strcmp(a.nombre, "fin") != 0) {
        anyadir_elemento(v, &a);
        leer_alumno_alternativo(&a);
    }
}
```

Cadenas de Caracteres Avanzadas

- Las cadenas de caracteres constantes (literales) también se pueden representar como **arrays sin tamaño** `const char nombre[]`.
  - En este caso, la variable contiene un array de caracteres que almacena la cadena de caracteres, donde el tamaño del array está determinado por la cantidad de caracteres (más el terminador *nulo*) de la cadena.
- Las cadenas de caracteres constantes (literales) también se pueden representar con el tipo `const char* nombre`.
  - En este caso, la variable contiene un puntero al primer carácter del array de caracteres que almacena la cadena de caracteres.

```
const char* AUTOR_1 = "Jose Luis";  // Puntero a Array interno constante de 10 caracteres
const char AUTOR_2[] = "Jose Luis"  // Array constante de 10 caracteres
const char AUTOR_3[] = { 'J', 'o', 's', 'e', ' ', 'L', 'u', 'i', 's', '\0' };

int main()
{
  const char* nombre_1 = "Jose Luis Lopez";    // puntero a cadena de caracteres constant
  char nombre_2[] = "Jose Luis Lopez";         // Array con 16 caracteres: Jose Luis Lope
  char nombre_3[] = { 'J', 'o', 's', 'e', ' ', 'L', 'u', 'i', 's', '\0' };
}
```

Cadenas de Caracteres Avanzadas. Paso de Parámetros

- En el caso del **paso de cadenas de caracteres constantes de entrada**, considerando que no serán modificadas, y que contienen el **caracter nulo terminador**, también se podrá realizar el paso de parámetros como **array sin tamaño** o como **puntero**.

```
// FORMA ALTERNATIVA-1 (parámetro array constante sin tamaño) (cantidad de caracteres determinada por \0)
void copiar_1(int maxcars, char dst[maxcars], const char org[])
{
    int i;
    for (i = 0; (i < maxcars-1)&&(org[i] != '\0'); ++i) {
        dst[i] = org[i];
    }
    dst[i] = '\0';
}
```

```
int main()
{
   const char texto1[] = "hola, adios";
   char texto2[MAXCARS];
   copiar_1(MAXCARS, texto2, texto1);
}
```

```
// FORMA ALTERNATIVA-2 (parámetro puntero a const char) (cantidad de caracteres determinada por \0)
void copiar_2(int maxcars, char dst[maxcars], const char* org)
{
    int i;
    for (i = 0; (i < maxcars-1)&&(org[i] != '\0'); ++i) {
        dst[i] = org[i];
    }
    dst[i] = '\0';
}
```

```
int main()
{
   const char* texto1 = "hola, adios";
   char texto2[MAXCARS];
   copiar_2(MAXCARS, texto2, texto1);
}
```

Cadenas de Caracteres Avanzadas. Operaciones

Salida a cadenas de caracteres

- `snprintf(destino, maxcars, "...formato...", ...);` vuelca la salida de texto al **array de char destino**, con un tamano máximo especificado.

```c
#include <stdio.h>
enum {
    MAXCARS = 127+1, // maximo 127 caracteres + 1 terminador
};
int main()
{
    char texto1[MAXCARS] = "Jose Luis";

    char texto2[MAXCARS];
    snprintf(texto2, MAXCARS, "Nombre: %s; Edad: %d\n", texto1, 20);

    printf("%s\n", texto2);     // muestra "Nombre: Jose Luis; Edad: 20" en pantalla
}
```

Cadenas de Caracteres Avanzadas. Operaciones

Entrada "segura" de cadenas de caracteres

```
void leer_palabra(int maxcars, char palabra[maxcars])
{
    char formato[20];
    snprintf(formato, 20, " %%%ds", maxcars-1);
    scanf(formato, palabra);                        // scanf(" %31s", palabra);
}

void leer_frase(int maxcars, char frase[maxcars])
{
    char formato[20];
    snprintf(formato, 20, " %%%d[^\n]", maxcars-1);
    scanf(formato, frase);                          // scanf(" %31[^\n]", frase);
}
```

Cadenas de Caracteres Avanzadas. Operaciones

Entrada desde cadenas de caracteres

- `sscanf(texto, "...formato...", ...);` extrae de la cadena de caracteres **texto** los datos de entrada, según el formato especificado.

```
int edad;
char texto2[MAXCARS] = "  Ana ; 20";
char palabra1[MAXCARS];
sscanf(texto2, " %127s ; %d", palabra1, &edad);

char texto3[MAXCARS] = "  Jose Luis ; 35";
char texto1[MAXCARS];
sscanf(texto3, " %127[^\n;] ; %d", texto1, &edad); // lee hasta ; o \n
```

Cadenas de Caracteres Avanzadas. Operaciones

Conversiones entre los tipos básicos y cadenas de caracteres

```c
#include <stdlib.h>

int    valor1 = atoi("1234");       // Conversión de cadena de caracteres a int
double valor2 = atof("123.456e7"); // Conversión de cadena de caracteres a double

sscanf("1234", " %d", &valor1);        // Conversión de cadena de caracteres a int
sscanf("123.456e7", " %lg", &valor2); // Conversión de cadena de caracteres a double

snprintf(destino, MAXCARS, "%d", 1234);        // Conversión de int a cadena de caracteres
snprintf(destino, MAXCARS, "%lg", 123.456e7); // Conversión de double a cadena de caracte
```

Definición de Constantes de Tipos Compuestos. Ejemplo 1

- Array constante de cadenas de caracteres.

```
enum {
    NELMS = 7,
    MAXCARS = 16,
};

const char* const DIAS_1[NELMS] = {
    "Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado", "Domingo"
};

const char DIAS_2[NELMS][MAXCARS] = {
    "Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado", "Domingo"
};
```

Definición de Constantes de Tipos Compuestos. Ejemplo 2

- Lista constante de números.

```
enum {
    NELMS = 20,
};
struct Lista {
    int nelms;
    int elm[NELMS];
};

const struct Lista DATOS = { 3, { 1, 2, 3 } }; // valor cero para el resto de elementos

int main()
{
    struct Lista datos = DATOS;
    // ...
}
```

Definición de Constantes de Tipos Compuestos. Ejemplo 3

- Array constante de personas.

```
enum {
    NELMS = 3,
    MAXCARS = 31+1,
};
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
struct Persona {
    char nombre[MAXCARS];
    struct Fecha fnac;
};
const struct Persona DATOS[NELMS] = {
    { "Lola", { 20, 4, 2010 } } ,
    { "Pepe", { 12, 8, 2011 } } ,
    { "Luis", { 24, 9, 2012 } }
};
```

Definición de Constantes de Tipos Compuestos. Ejemplo 4

- Lista constante de personas.

```
enum {
    NELMS = 20,
    MAXCARS = 31+1,
};
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
struct Persona {
    char nombre[MAXCARS];
    struct Fecha fnac;
};
struct Lista {
    int nelms;
    struct Persona elm[NELMS];
};
const struct Lista DATOS = { 3,
    {
        { "Lola", { 20, 4, 2010 } } ,
        { "Pepe", { 12, 8, 2011 } } ,
        { "Luis", { 24, 9, 2012 } }
    }
};
int main()
{
    struct Lista datos = DATOS;
}
```