# U.D.3

# Estructuras de almacenamiento

***(PARTE 1: VECTORES)***

**Nota:** Ejemplos en lenguaje C

# ÍNDICE

# 1. Estructuración de datos

- Datos simples: constantes, tipos básicos (variables).
- Necesidad: agrupar los datos de un programa ➔ estructura de datos.
- El tamaño de una estructura de datos es la cantidad de memoria que ocupa.

- Tipos:
  - Estáticas: Su tamaño no varía en tiempo de ejecución.
  - Dinámicas: Su tamaño puede variar en tiempo de ejecución.

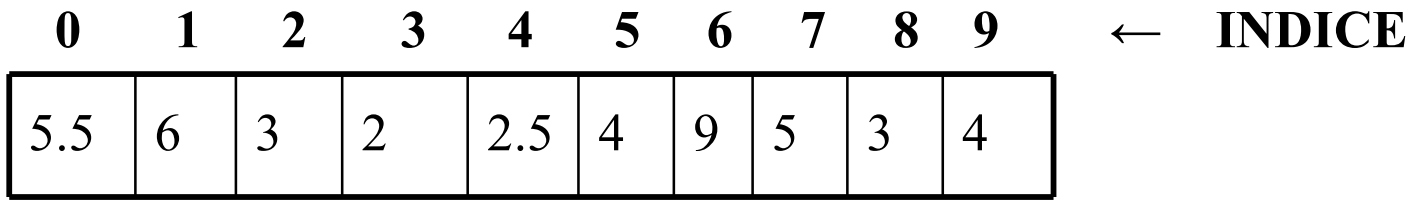
# Tipos de Estructuras de Datos

- Estáticas
  - *Arrays* o vectores
  - Tablas o matrices
  - Registros
- Dinámicas
  - Listas
  - Pilas y colas
  - Árboles

# 2. Concepto de *array* (vector)

- Colección de elementos del mismo tipo, donde cada elemento está asociado a un índice (posición que ocupa).
- Queda determinado por:
  - Número de elementos.
  - Tipo de sus elementos.

```
float notas[10];
```

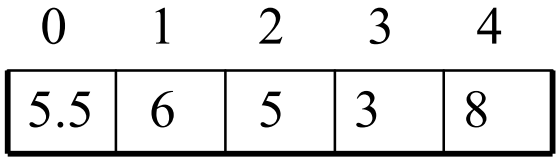


# *Arrays*

- Definición:

```
float  nota[5];
```

- Índices (0 … 4):




- Acceso a la nota del alumno 4:

```
printf("%f", nota[3]);
```

# Elementos que caracterizan un *array*

- Es un conjunto de datos del mismo tipo.
- Se identifica por su nombre.
- Es un tipo de dato estructurado y estático.
- Sus componentes individuales se llaman elementos y están almacenados en la memoria en posiciones contiguas.
  - A ellos se puede acceder directamente mediante el uso de subíndices que indican su posición.
- Para poder utilizarlo hay que indicar el tipo de dato que se va a almacenar en cada posición y el nº de posiciones reservadas.

# Inicialización de un vector en la declaración

- Sintaxis:

```
tipo nombre []={valor 1, valor 2...}
```

Ejemplos:

```
int vector[]={1,2,3,4,5,6,7,8};
char vector[]={'p','r','o','g','r','a','m','a','d','o','r'};
```

- Si lo hacemos así no es necesario indicar el tamaño.

- C no comprueba los límites de los *arrays*.
  - Esto quiere decir que si hacemos v[20] para un *array* de 10 posiciones *array* anterior, el C no nos va a informar de ningún error. Es responsabilidad del programador el indexamiento correcto de un *array*.

# Operaciones con vectores

- Operaciones con elementos:
  - Asignación.
  - Lectura.
  - Escritura.
- Operaciones sobre el vector:
  - Recorrido.
  - Búsqueda.
  - Ordenación.
  - Inserción.
  - Eliminación.

# 3. Recorrido de un array

- Recorrer un *array* es realizar un tratamiento a cada uno de los componentes del *array*.
- El programador debe preocuparse de que el índice no se pase de los límites del vector.
- Inicialización de un *array*:

```
for (i=0; i<NALUMNOS; i++)
{
    notas[i] = 0;
}
```

# Recorrido de *arrays*

- Leer los datos y almacenarlos en el array:

```
for (i=0; i<NALUMNOS; i++)
{
   printf("Nota alumno %d", i+1);
   scanf("%f", &nota[i]);
   fflush(stdin);
}
```

- Imprimir el contenido del vector:

```
for (i=0; i<NALUMNOS; i++)
{
   printf("Nota alumno %d es %f", i+1, nota[i]);
}
```

# 4. Paso de *arrays* como argumento a una función

- Un vector puede pasarse como argumento a una función.

- Una función nunca puede devolver un vector.

- Los vectores siempre son pasados por referencia, es decir, cualquier cambio realizado en los elementos del vector será visible al salir de la función.

# *Arrays* como parámetros

```
#define TAM 10
void inicializar( int [ ] )
void main()
{
   int v[TAM];
   inicializar (v);
   ……
}

void inicializar (int v[ ])
{
   ……
}
```

# *Arrays* como parámetros: Cabeceras

La cabecera de la función podría ser de estas 3 formas:

- `void inicializar (int v[TAM])`
- `void inicializar (int v[ ])`
- `void inicializar (int *v)`

Importante: el nombre del vector sin ningún índice es un puntero al principio del vector.

# 5.Búsqueda de un elemento en un vector

- La finalidad es saber si un elemento está o no en un vector. Además si está debe informarnos de su posición.
- Argumentos de entrada:
  - `num` es el elemento que quiero buscar.
  - `v` es un vector de elementos (en este caso enteros).
  - `tam` es el tamaño del vector.
- Devuelve: `-1` si el elemento no está en el vector y la posición en la que se encuentra en caso contrario.

# Algoritmo de búsqueda

```
int busca_valor (int n, int v[], int tam)
{
   int i = 0, pos = -1;

   while (i < tam && pos == -1)
   {
        if (v[i] == n)
            pos = i;
        i++;
   }

   return pos;
}
```

# Búsqueda en un vector ordenado

- La finalidad del algoritmo es la misma, pero queremos encontrar una solución más eficiente en el caso de que el vector de elementos esté ordenado (supongamos de forma ascendente).
- Dos soluciones:
  1. Algoritmo de búsqueda ordenada.
  2. Algoritmo de búsqueda binaria.

# Algoritmo de búsqueda ordenada

```
int busqueda_sec_ordenada (int num, int v[], int tam)
{
  int i, pos;

  i = 0;
  pos = -1;

  while ((i < tam) && (pos == -1) && (v[i] <= num))
  {
    if (v[i] == num)
      pos = i;
    i++;
  }

  return pos;
}
```

# Algoritmo de búsqueda binaria

- Para vectores ordenados.
- Se basa en comprobar si el dato es mayor o menor que el dato central del vector y repetir la búsqueda en la mitad correspondiente.
- La condición de finalización es que encontremos el dato o que el vector no se pueda dividir más.

```
int busqueda_binaria (int num, int v[], int tam)
{
  int izq, der, centro;
  int encontrado;

  izq = 0;
  der = tam-1;
  encontrado = 0; // Falso

  while ((izq <= der) && (!encontrado))
  {
    centro = (izq + der) / 2;
    if (v[centro] == num)
      encontrado = 1; // Verdadero
    else
      if (num > v[centro])
        izq = centro + 1;
      else
        der = centro - 1;
  }

  if (!encontrado)
    centro = -1;  // No lo ha encontrado, posición "ficticia".

  return centro;
}
```

# 6.Algoritmos de ordenación de un vector

La finalidad es ordenar (de forma creciente o decreciente ) los elementos de un vector.

## 6.1. Algoritmo de selección

- Se trata de encontrar el elemento mas pequeño del vector e intercambiarlo con el de la primera casilla. Este proceso se repite tomando en cada vuelta una parte más pequeña del vector.

# Algoritmo de ordenación por selección

```
void ordena_seleccion (int v[], int tam)
{
  int i, j, menor;
  int aux; // Variable para intercambio.

  for (i = 0; i < tam; i++)
  {
    menor = i;  // Se calcula el menor elemento a partir de i.
    for (j = i+1; j < tam; j++)
      if (v[j] < v[menor])
        menor = j;

    // Se intercambia el elemento i con el elemento menor.
    aux = v[i];
    v[i] = v[menor];
    v[menor] = aux;
  }
}
```

# 6.2.Algoritmo de ordenación de la burbuja

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados.

```
void ordena_burbuja (int v[], int tam)
{
  int i, j, aux;

  for (i = 0; i < tam; i++)
    for (j = tam-1; j > i; j--)
      if (v[j] < v[j-1])
      {
        aux = v[j];
        v[j] = v[j-1];
        v[j-1] = aux;
      }
}
```

# 7. Inserción (y borrado)

- Se selecciona en qué posición se insertará el elemento nuevo:
  - Si el vector está desordenado, deberá indicarse.
  - Si el vector está ordenado, deberá obtenerse.
- Se desplazan las celdas a la derecha de la posición una posición hacia delante.
  - Deberá realizarse de derecha a izquierda para no perder los valores.
  - Si el vector inicial está lleno:
    - no se podrá realizar la inserción, o bien
    - se perderá el contenido de la última celda.
- Se inserta el nuevo valor en la posición.

# 7. (Inserción y) borrado

- Puede hacerse con o sin dejar huecos.
- Se selecciona qué posición deberá borrarse.
- Dos posibilidades:
  - Se “marca” la celda como borrada (huecos)
  - Se desplazan las celdas a la derecha de la posición una posición hacia atrás y se “marca” la última celda como “libre” (sin huecos).