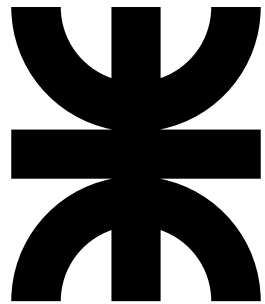
Informática II
Relación entre punteros y arreglos

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Repaso sobre arreglos

Definición de arreglo

- Estructura de dato que consiste en elementos relacionados del mismo tipo
- Colección de datos bajo un mismo nombre

Repaso sobre arreglos

Definición de arreglo

- Estructura de dato que consiste en elementos relacionados del mismo tipo
- Colección de datos bajo un mismo nombre

Desde un punto de vista de bajo nivel:

- Es un grupo de posiciones de memoria consecutivas relacionadas entre sí
- Todas del mismo nombre y del mismo tipo

Repaso sobre arreglos

Definición de arreglo

- Estructura de dato que consiste en elementos relacionados del mismo tipo
- Colección de datos bajo un mismo nombre

Desde un punto de vista de bajo nivel:

- Es un grupo de posiciones de memoria consecutivas relacionadas entre sí
- Todas del mismo nombre y del mismo tipo

Por ejemplo:

```
int a = 10; /* Definición de un entero */
int v[3]; /* Definición de un arreglo de enteros */
```

Repaso sobre arreglos

Definición de arreglo

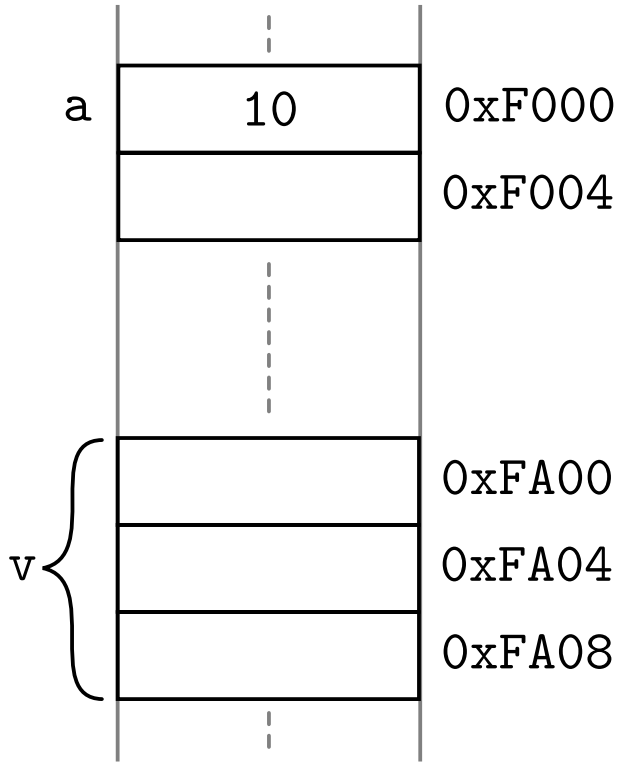
- Estructura de dato que consiste en elementos relacionados del mismo tipo
- Colección de datos bajo un mismo nombre

Desde un punto de vista de bajo nivel:

- Es un grupo de posiciones de memoria consecutivas relacionadas entre sí
- Todas del mismo nombre y del mismo tipo

Por ejemplo:

```
int a = 10; /* Definición de un entero */
int v[3]; /* Definición de un arreglo de enteros */
```



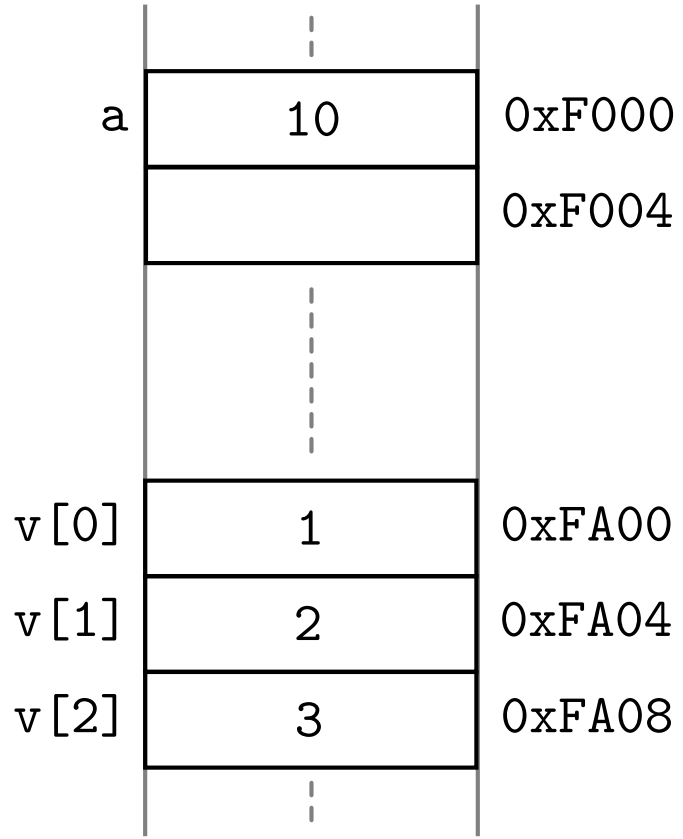
Repaso sobre arreglos – ejemplo

- ¿Cómo se inicializa un arreglo?
- ¿Cómo acceder a un elemento particular?

Repaso sobre arreglos – ejemplo

- ¿Cómo se inicializa un arreglo?
- ¿Cómo acceder a un elemento particular?

```c
#include <stdio.h>

#define TAM 3

int main(void)
{
  int a = 10;
  int v[TAM] = {1, 2, 3}; /* arreglo */
  int i; /* subíndice */

  for(i = 0; i < TAM; i++)
    printf("%d\n", v[i]);

  return 0;
}
```



Punteros

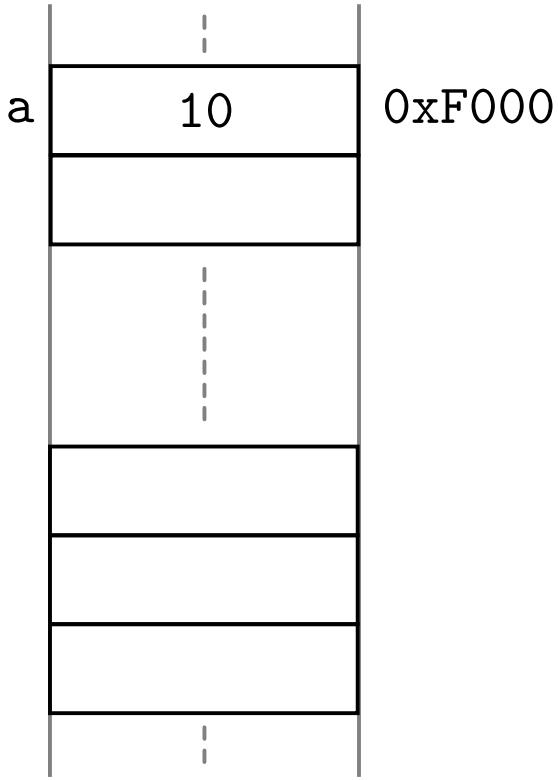
Definición

Son variables cuyos valores representan posiciones de memoria

Punteros

Definición

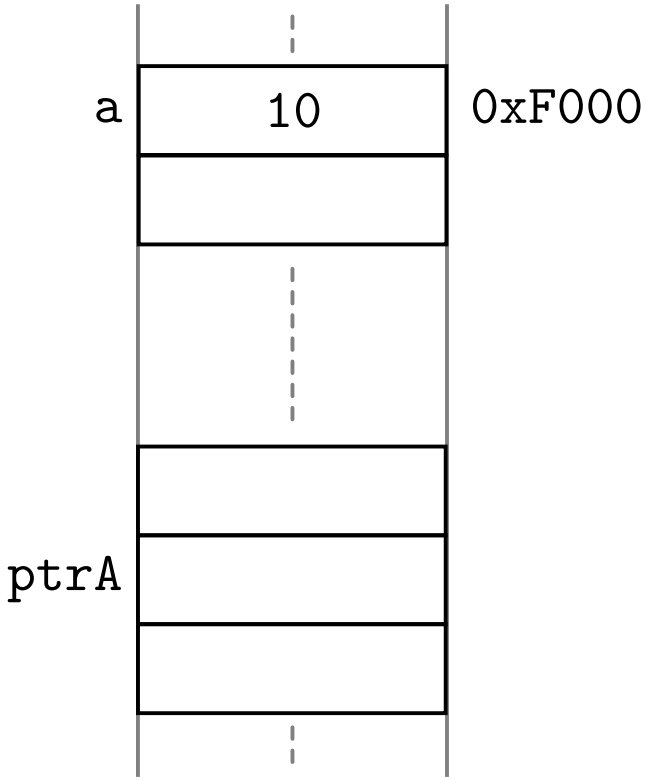
Son variables cuyos valores representan posiciones de memoria



Punteros

Definición

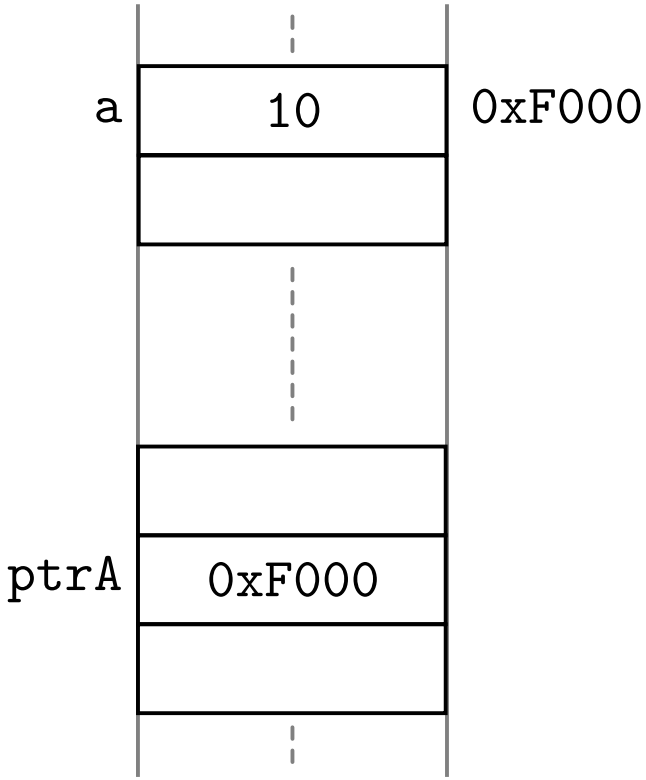
Son variables cuyos valores representan posiciones de memoria



Punteros

Definición

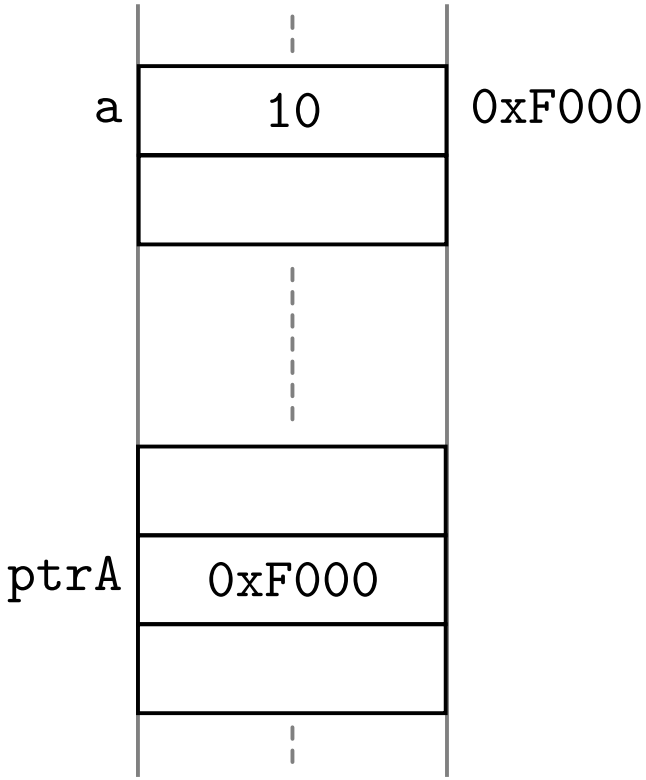
Son variables cuyos valores representan posiciones de memoria



Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



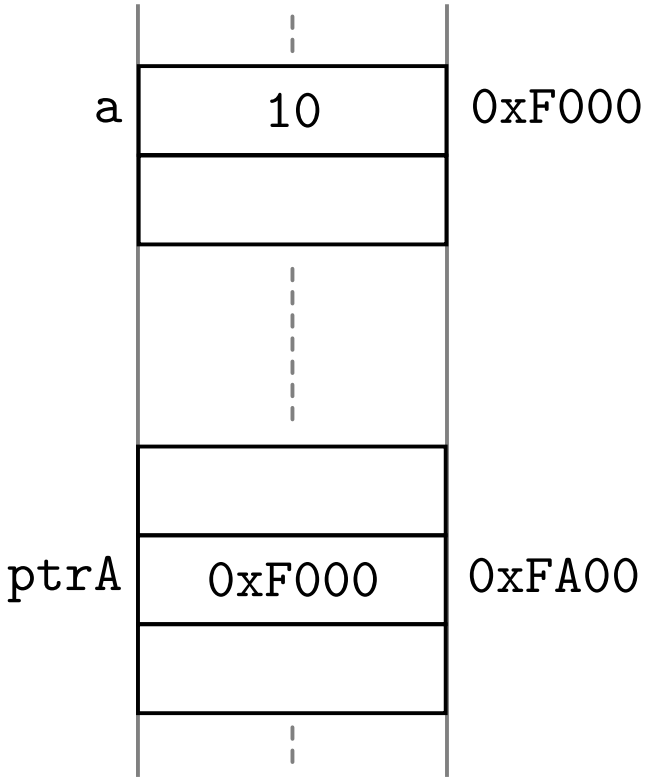
Por ejemplo:

```
int a = 10; /* Definición de un entero */
int *ptrA = &a; /* Definición de un puntero */
```

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



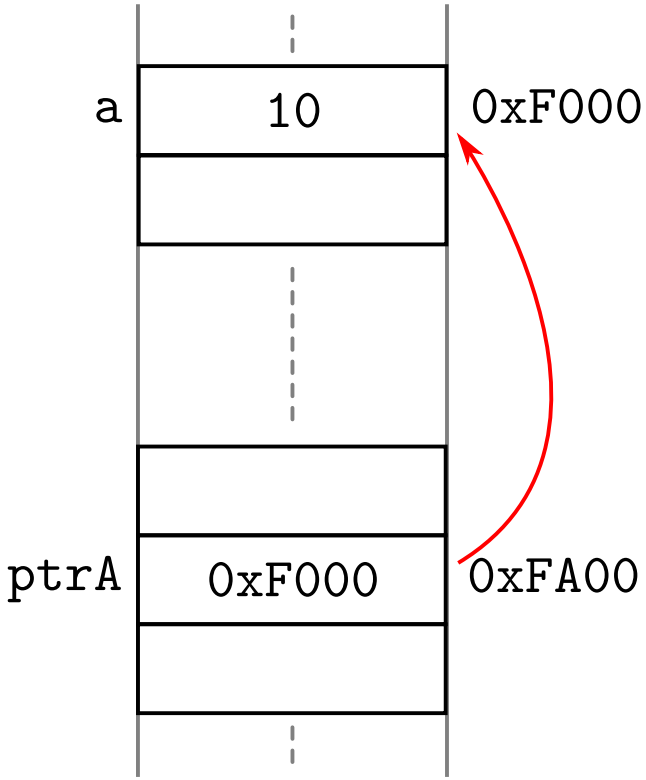
Por ejemplo:

```
int a = 10; /* Definición de un entero */
int *ptrA = &a; /* Definición de un puntero */
```

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



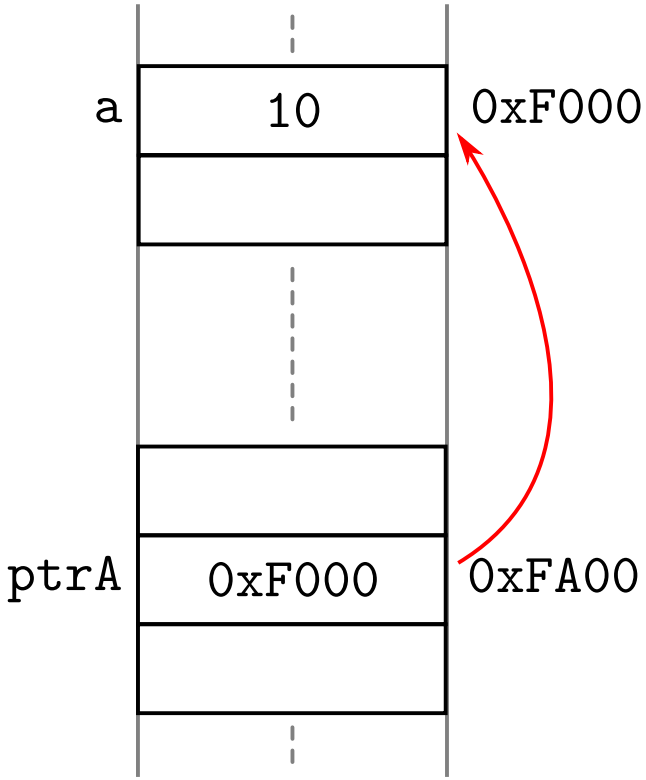
Por ejemplo:

```
int a = 10; /* Definición de un entero */
int *ptrA = &a; /* Definición de un puntero */
```

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



Por ejemplo:

```
int a = 10; /* Definición de un entero */
int *ptrA = &a; /* Definición de un puntero */
```

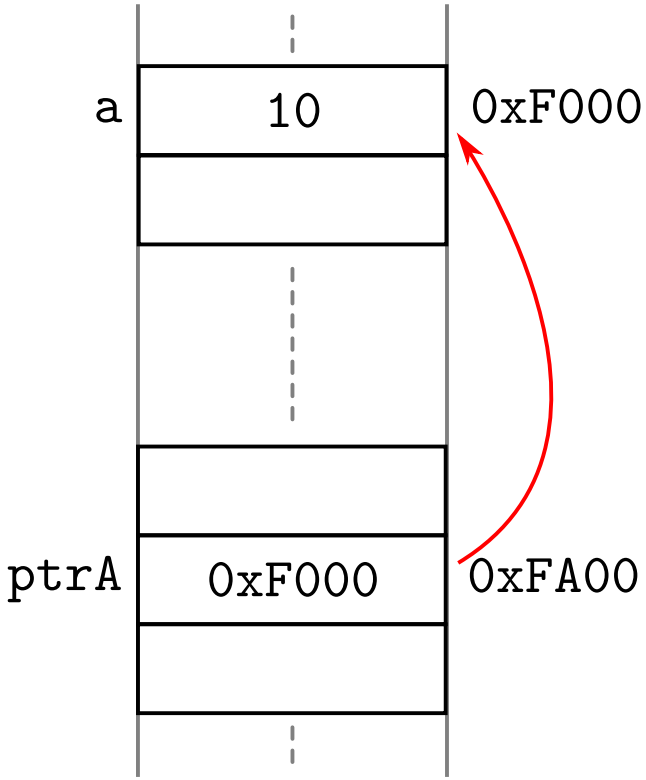
O sea:

- Una variable hace referencia a un valor de *forma directa*

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



Por ejemplo:

```
int a = 10; /* Definición de un entero */
int *ptrA = &a; /* Definición de un puntero */
```

O sea:

- Una variable hace referencia a un valor de *forma directa*
- Un puntero hace referencia a un valor de *forma indirecta*

Punteros – ejemplo

```c
#include <stdio.h>

int main(void) {
  int a; /* variable tipo int */
  int *ptrA; /* puntero a variable tipo int */









  return 0;
}
```

Punteros – ejemplo

```c
#include <stdio.h>

int main(void) {
  int a; /* variable tipo int */
  int *ptrA; /* puntero a variable tipo int */

  a = 10;
  ptrA = &a; /* ptrA toma la dirección de a */




  return 0;
}
```

Punteros – ejemplo

```c
#include <stdio.h>

int main(void) {
  int a; /* variable tipo int */
  int *ptrA; /* puntero a variable tipo int */

  a = 10;
  ptrA = &a; /* ptrA toma la dirección de a */

  printf("La dirección de a es %p\n", &a);
  printf("El valor de ptrA es %p\n", ptrA);



  return 0;
}
```

Punteros – ejemplo

```c
#include <stdio.h>

int main(void) {
  int a; /* variable tipo int */
  int *ptrA; /* puntero a variable tipo int */

  a = 10;
  ptrA = &a; /* ptrA toma la dirección de a */

  printf("La dirección de a es %p\n", &a);
  printf("El valor de ptrA es %p\n", ptrA);



  return 0;
}
```

```
La dirección de a es 0x7ffe4271501c
El valor de &a es 0x7ffe4271501c
```

Punteros – ejemplo

```c
#include <stdio.h>

int main(void) {
  int a; /* variable tipo int */
  int *ptrA; /* puntero a variable tipo int */

  a = 10;
  ptrA = &a; /* ptrA toma la dirección de a */

  printf("La dirección de a es %p\n", &a);
  printf("El valor de ptrA es %p\n", ptrA);

  printf("\nEl valor de a es %d\n", a);
  printf("El valor de *ptrA es %d\n", *ptrA);
  return 0;
}
```

```
La dirección de a es 0x7ffe4271501c
El valor de &a es 0x7ffe4271501c
```

Punteros – ejemplo

```c
#include <stdio.h>

int main(void) {
  int a; /* variable tipo int */
  int *ptrA; /* puntero a variable tipo int */

  a = 10;
  ptrA = &a; /* ptrA toma la dirección de a */

  printf("La dirección de a es %p\n", &a);
  printf("El valor de ptrA es %p\n", ptrA);

  printf("\nEl valor de a es %d\n", a);
  printf("El valor de *ptrA es %d\n", *ptrA);
  return 0;
}
```

```
La dirección de a es 0x7ffe4271501c
El valor de &a es 0x7ffe4271501c

El valor de a es 10
El valor de *ptrA es 10
```

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- Los punteros puede apuntar a objetos de cualquier tipo

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- Los punteros puede apuntar a objetos de cualquier tipo
- Se deben inicializar en la declaración o mediante una asignación

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- Los punteros puede apuntar a objetos de cualquier tipo
- Se deben inicializar en la declaración o mediante una asignación
- Se puede inicializar a `NULL` o a una dirección válida

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- Los punteros puede apuntar a objetos de cualquier tipo
- Se deben inicializar en la declaración o mediante una asignación
- Se puede inicializar a `NULL` o a una dirección válida
- Un puntero a `NULL` no apunta a ningún lado

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- Los punteros puede apuntar a objetos de cualquier tipo
- Se deben inicializar en la declaración o mediante una asignación
- Se puede inicializar a `NULL` o a una dirección válida
- Un puntero a `NULL` no apunta a ningún lado

Los *punteros* son una de las características más poderosas del lenguaje C.

Más sobre punteros

Operadores

- `&`: operador de dirección u operador de referencia
- `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- Los punteros puede apuntar a objetos de cualquier tipo
- Se deben inicializar en la declaración o mediante una asignación
- Se puede inicializar a `NULL` o a una dirección válida
- Un puntero a `NULL` no apunta a ningún lado

Los *punteros* son una de las características más poderosas del lenguaje C.

Sirven para:

- Simular llamadas de funciones por referencias
- Crear y manipular estructuras de datos dinámicas

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```c
#include <stdio.h>

int main(void)
{
  int a = 10;


  printf("Tamaño de a: %lu\n", sizeof a);



  return 0;
}
```

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```c
#include <stdio.h>

int main(void)
{
  int a = 10;
  int v[] = {1, 2, 3, 4, 5};

  printf("Tamaño de a: %lu\n", sizeof a);
  printf("Tamaño del vector: %lu\n", sizeof v);


  return 0;
}
```

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```c
#include <stdio.h>

int main(void)
{
  int a = 10;
  int v[] = {1, 2, 3, 4, 5};

  printf("Tamaño de a: %lu\n", sizeof a);
  printf("Tamaño del vector: %lu\n", sizeof v);
  printf("Tamaño del tipo 'double': %lu\n", sizeof(double));

  return 0;
}
```

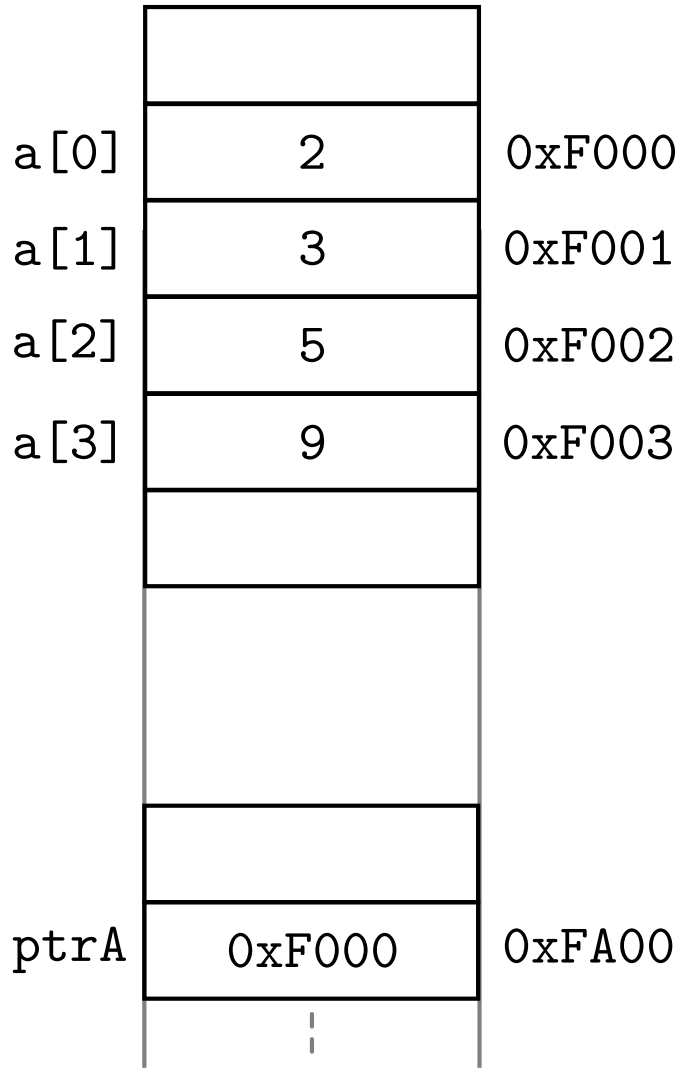
El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```c
#include <stdio.h>

int main(void)
{
  int a = 10;
  int v[] = {1, 2, 3, 4, 5};

  printf("Tamaño de a: %lu\n", sizeof a);
  printf("Tamaño del vector: %lu\n", sizeof v);
  printf("Tamaño del tipo 'double': %lu\n", sizeof(double));

  return 0;
}
```

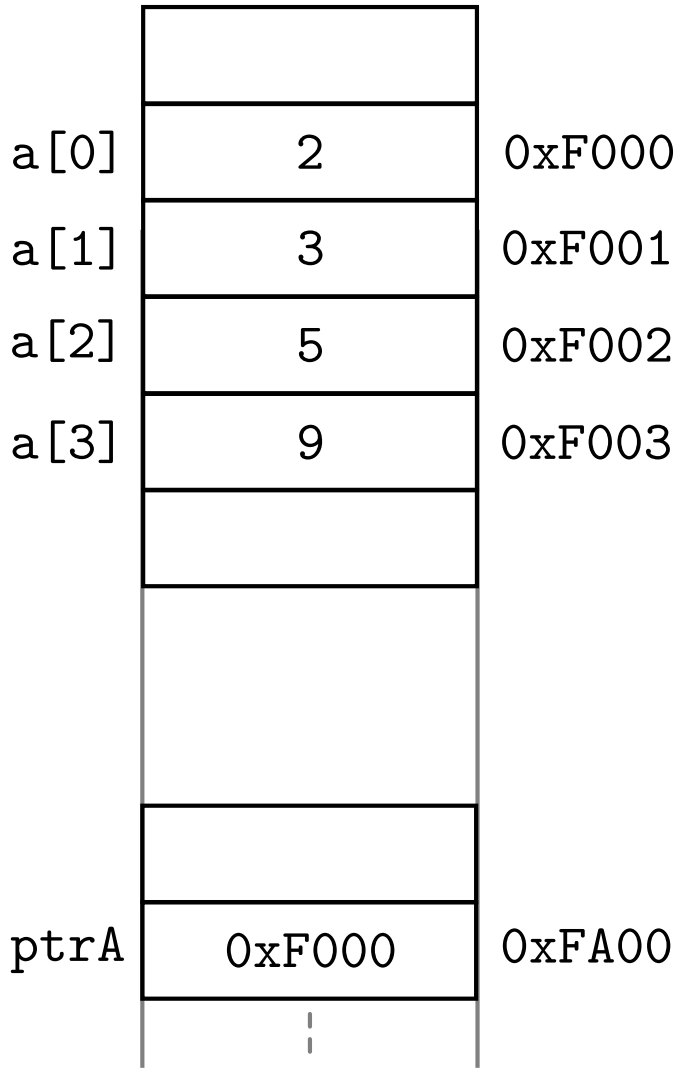
```
Tamaño de a: 4
Tamaño del vector: 20
Tamaño del tipo 'double': 8
```

Aritmética sobre punteros



Aritmética sobre punteros

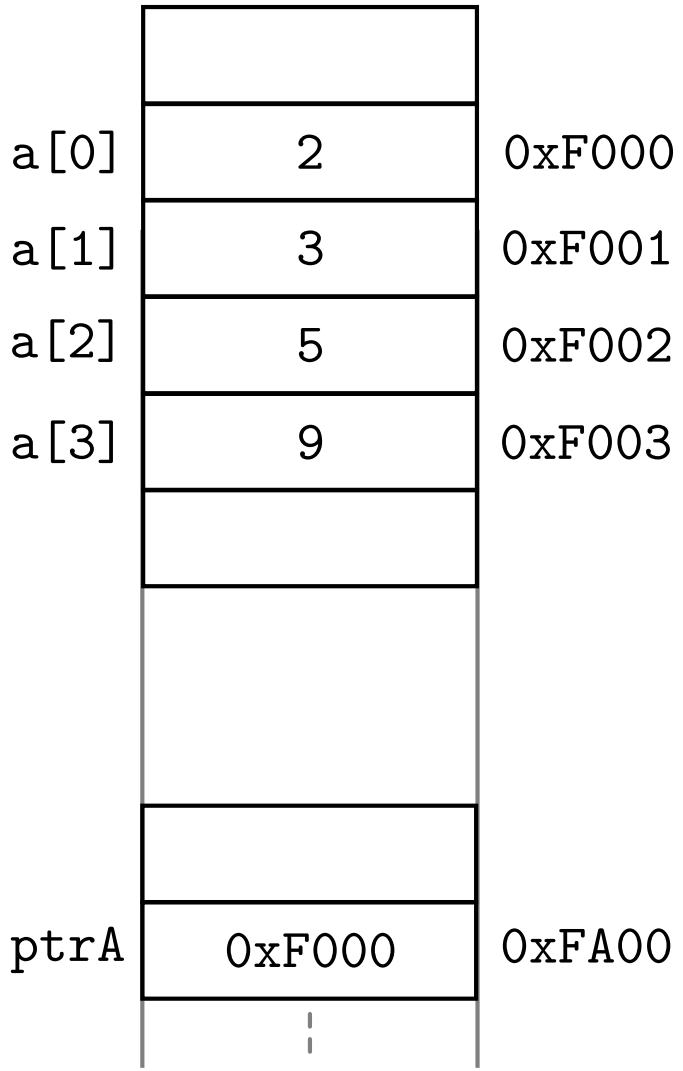
```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];


  printf("%d\n", *ptrA);

  return 0;
}
```



Aritmética sobre punteros

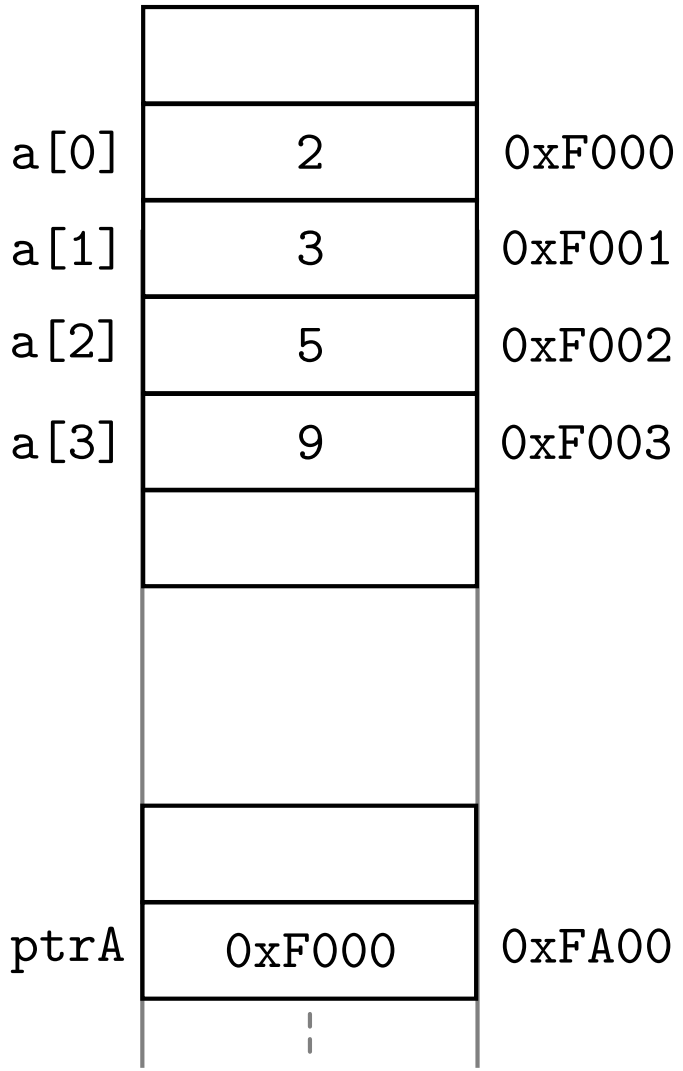
```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];


  printf("%d\n", *ptrA);

  return 0;
}
```

```
2
```



Aritmética sobre punteros

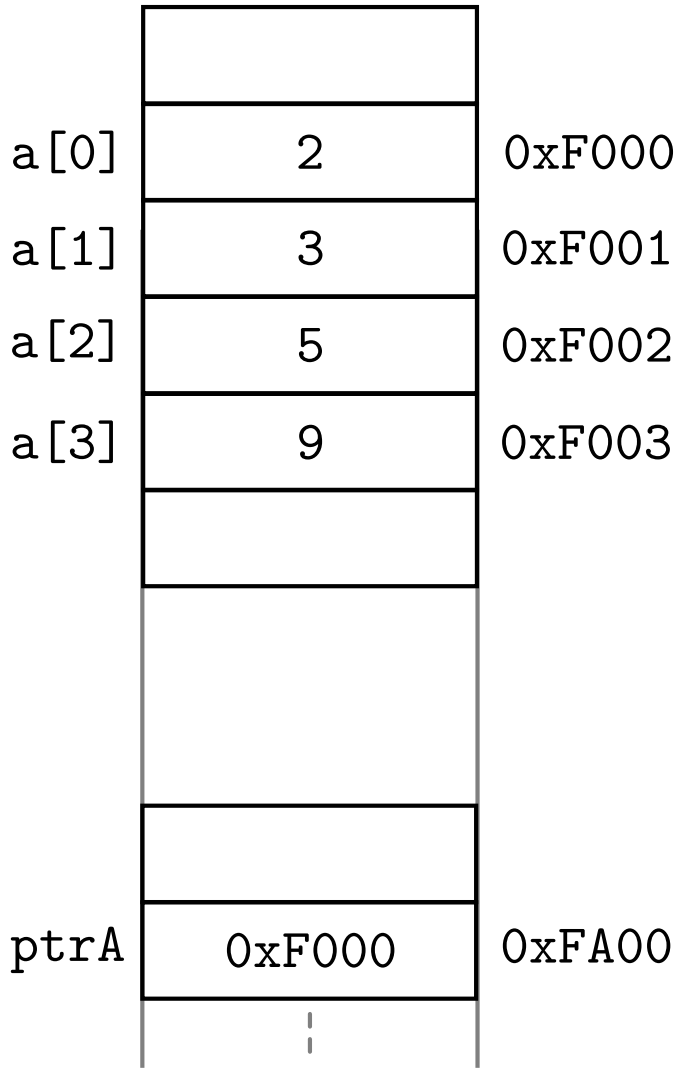
```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];
  ptrA = ptrA + 1; /* ptrA += 1 */

  printf("%d\n", *ptrA);

  return 0;
}
```



Aritmética sobre punteros

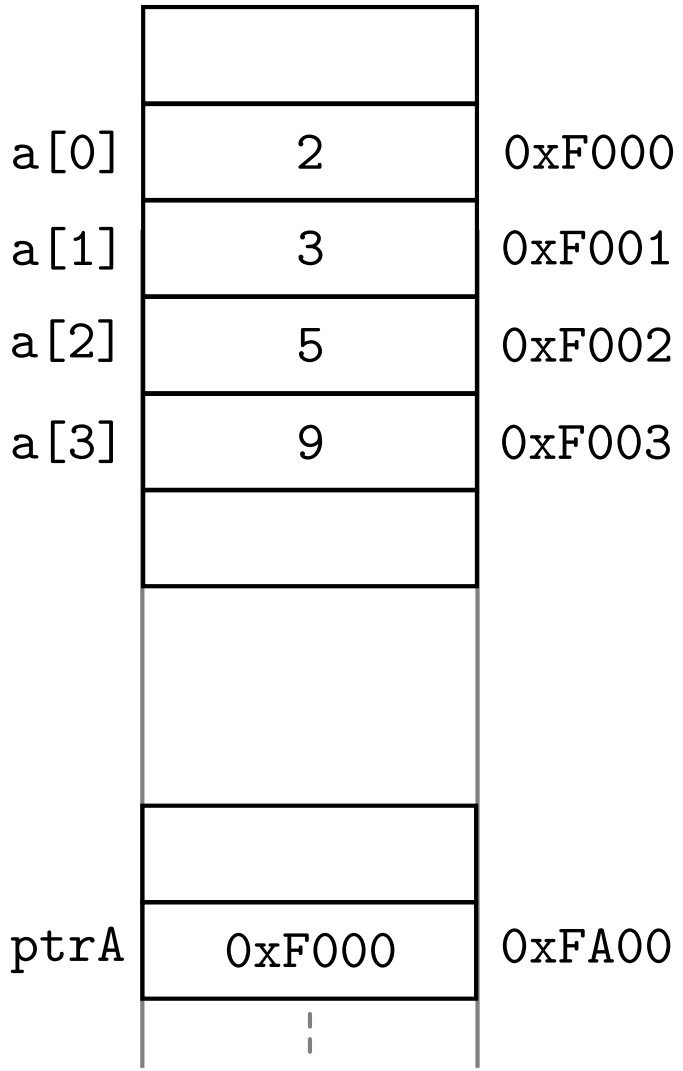
```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];
  ptrA = ptrA + 1; /* ptrA += 1 */

  printf("%d\n", *ptrA);

  return 0;
}
```

```
3
```



Aritmética sobre punteros

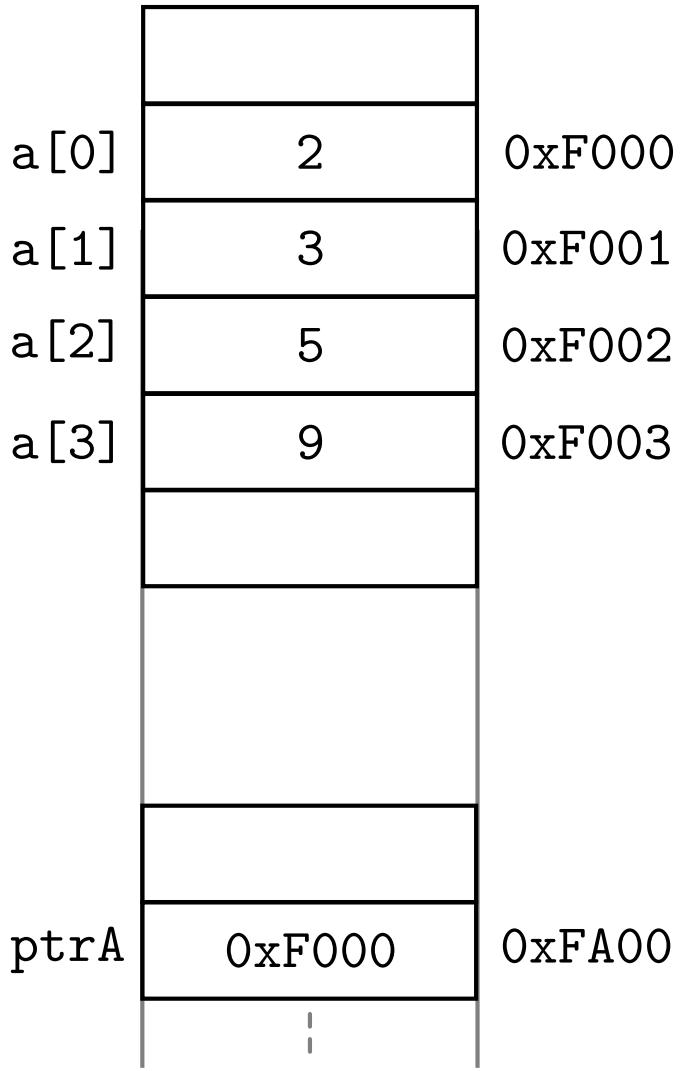
```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];


  printf("%d\n", *(ptrA + 2));

  return 0;
}
```



Aritmética sobre punteros

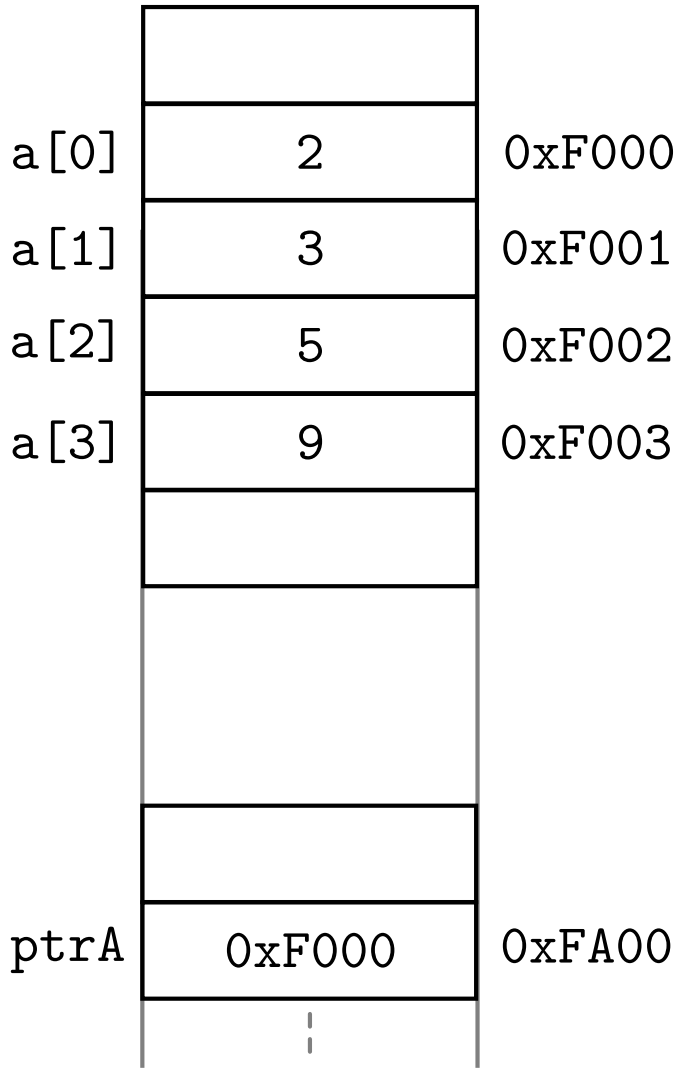
```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];


  printf("%d\n", *(ptrA + 2));

  return 0;
}
```

```
5
```



Aritmética sobre punteros

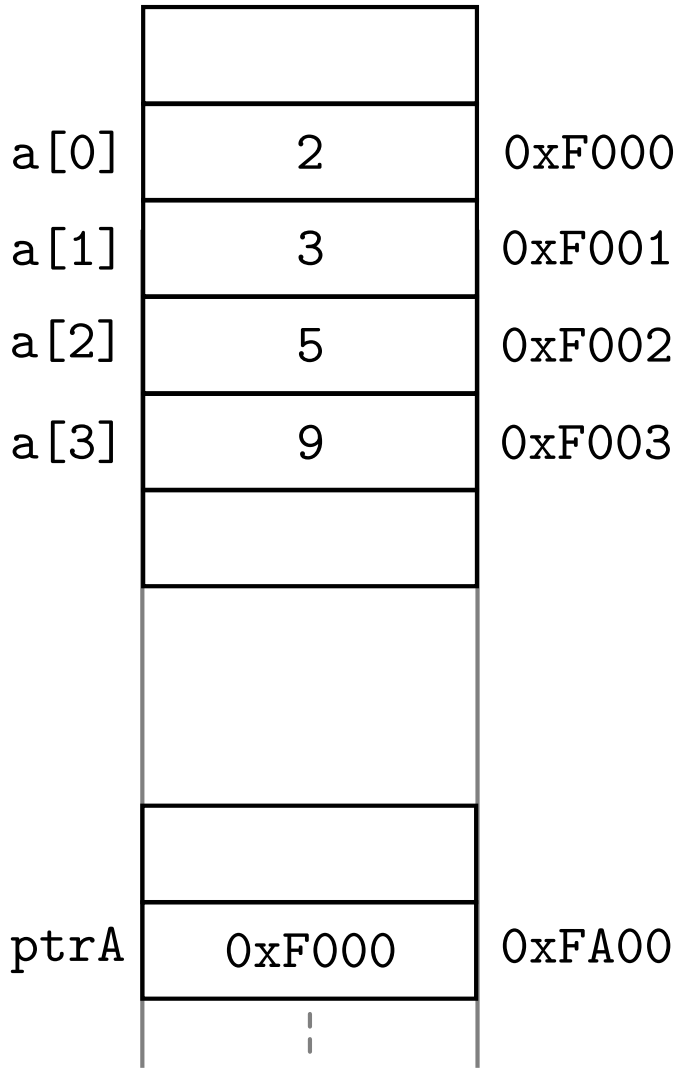
```
1  #include <stdio.h>
2  #define TAM 4
3
4  int main(void)
5  {
6    char a[TAM] = {2, 3, 5, 9};
7    char *ptrA;
8
9    ptrA = &a[0];
10
11
12   printf("%d\n", *ptrA + 2);
13
14   return 0;
15 }
```



Aritmética sobre punteros

```c
#include <stdio.h>
#define TAM 4

int main(void)
{
  char a[TAM] = {2, 3, 5, 9};
  char *ptrA;

  ptrA = &a[0];


  printf("%d\n", *ptrA + 2);

  return 0;
}
```

```
4
```



Aritmética sobre punteros

```c
#include <stdio.h>
#define TAM 8

int main(void)
{
  int i; /* índice */
  char v[TAM] = {0};
  char *ptrV = &v[0];

  for(i = 0; i < TAM; i++)
    printf("La dirección del elemento %d es %p\n", i, (ptrV + i));

  return 0;
}
```

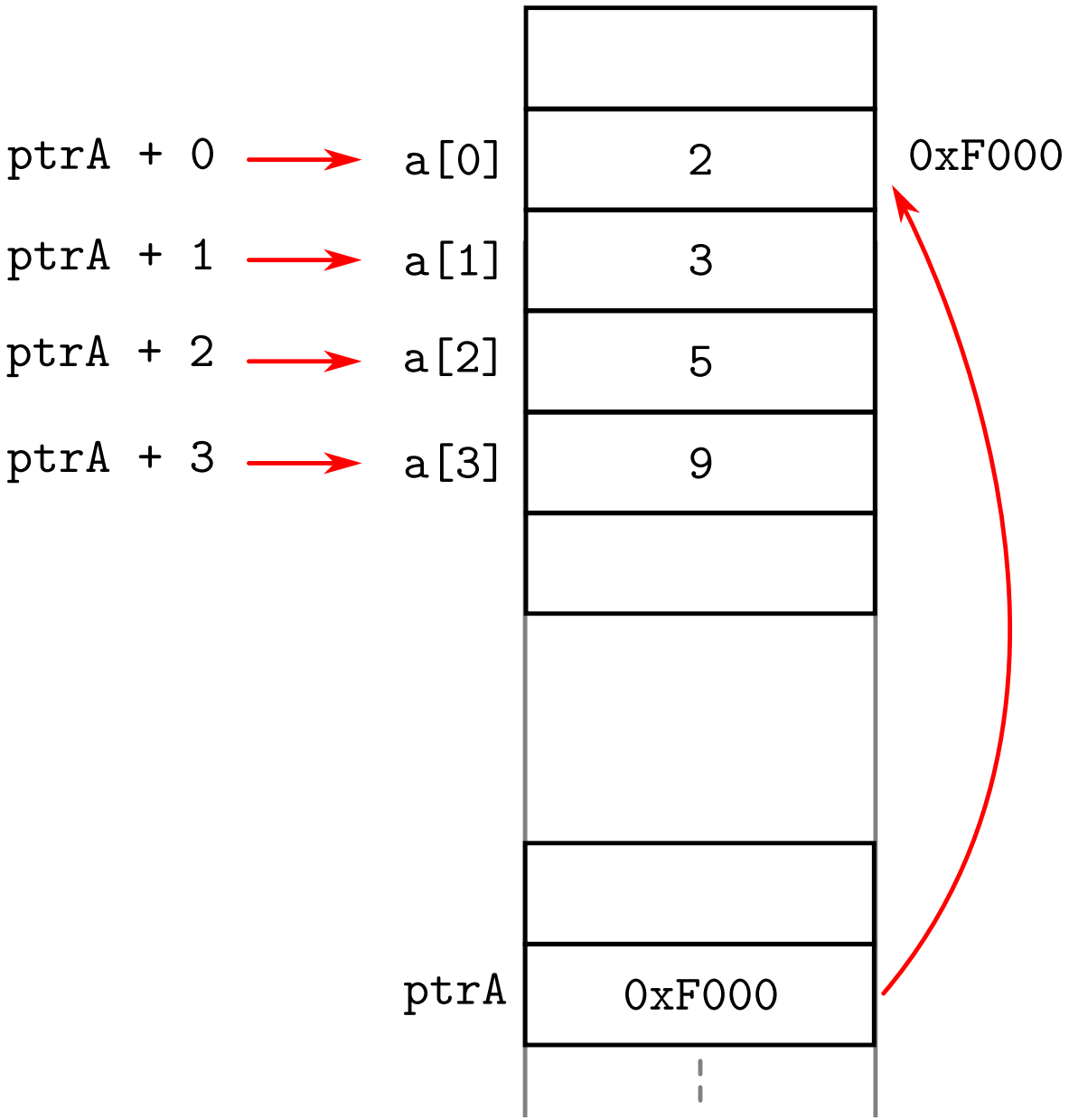
```
La dirección del elemento 0 es 0x7ffccbf01aa0
La dirección del elemento 1 es 0x7ffccbf01aa1
La dirección del elemento 2 es 0x7ffccbf01aa2
La dirección del elemento 3 es 0x7ffccbf01aa3
La dirección del elemento 4 es 0x7ffccbf01aa4
La dirección del elemento 5 es 0x7ffccbf01aa5
La dirección del elemento 6 es 0x7ffccbf01aa6
La dirección del elemento 7 es 0x7ffccbf01aa7
```

Aritmética sobre punteros

```c
#include <stdio.h>
#define TAM 8

int main(void)
{
  int i; /* índice */
  int v[TAM] = {0};
  int *ptrV = &v[0];

  for(i = 0; i < TAM; i++)
    printf("La dirección del elemento %d es %p\n", i, (ptrV + i));

  return 0;
}
```

```
La dirección del elemento 0 es 0x7ffe227a2710
La dirección del elemento 1 es 0x7ffe227a2714
La dirección del elemento 2 es 0x7ffe227a2718
La dirección del elemento 3 es 0x7ffe227a271c
La dirección del elemento 4 es 0x7ffe227a2720
La dirección del elemento 5 es 0x7ffe227a2724
La dirección del elemento 6 es 0x7ffe227a2728
La dirección del elemento 7 es 0x7ffe227a272c
```

Aritmética sobre punteros



Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta
- Un nombre de arreglo puede interpretarse como un puntero constante

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta
- Un nombre de arreglo puede interpretarse como un puntero constante

Por ejemplo:

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

int *ptrA = &a[0];
```

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta
- Un nombre de arreglo puede interpretarse como un puntero constante

Por ejemplo:

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

int *ptrA = &a[0];
```

Es equivalente a:

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

int *ptrA = a;
```

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta
- Un nombre de arreglo puede interpretarse como un puntero constante
- Se puede utilizar los punteros para operaciones con subíndices de arreglos

Por ejemplo:

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

int *ptrA = &a[0];
```

Es equivalente a:

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

int *ptrA = a;
```

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta
- Un nombre de arreglo puede interpretarse como un puntero constante
- Se puede utilizar los punteros para operaciones con subíndices de arreglos

Por ejemplo:

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

int *ptrA = &a[0];
```

```
printf("%d", a[3]);

printf("%d", *(ptrA + 3));
```

Es equivalente a:

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

int *ptrA = a;
```

Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados
- En algunos casos se pueden utilizar de manera indistinta
- Un nombre de arreglo puede interpretarse como un puntero constante
- Se puede utilizar los punteros para operaciones con subíndices de arreglos

Por ejemplo:

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

int *ptrA = &a[0];
```

```
printf("%d", a[3]);

printf("%d", *(ptrA + 3));
```

Es equivalente a:

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

int *ptrA = a;
```

```
printf("%d", ptrA[3]);

printf("%d", *(a + 3));
```

Relación entre punteros y arreglos

```c
#include <stdio.h>

int main(void)
{
  int a[] = {1, 2, 3, 4, 5};
  int *ptrA = a;

  printf("a[3] : %d\n", a[3]);
  printf("*(ptrA + 3) : %d\n", *(ptrA + 3));
  printf("ptrA[3] : %d\n", ptrA[3]);
  printf("*(a + 3) : %d\n", *(a + 3));

  return 0;
}
```

```
a[3]       : 4
*(ptrA+3)  : 4
ptrA[3]    : 4
*(ptrA+3)  : 4
```

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

`*(ptrA+i)`: Notación puntero/desplazamiento

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

`*(ptrA+i)`: Notación puntero/desplazamiento

`ptrA[i]`: Notación puntero/subíndice

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

`*(ptrA+i)`: Notación puntero/desplazamiento

`ptrA[i]`: Notación puntero/subíndice

`*(a+i)`: Notación puntero/desplazamiento donde el puntero es el nombre del arreglo

Ejemplo: tamaño de un arreglo/puntero

```c
#include <stdio.h>
#define TAM 20

void imprimirTamanio(float v[])
{
  printf("Tamaño del arreglo: %lu\n", sizeof(v));
}

int main(void)
{









  return 0;
}
```

Ejemplo: tamaño de un arreglo/puntero

```c
#include <stdio.h>
#define TAM 20

void imprimirTamanio(float v[])
{
  printf("Tamaño del arreglo: %lu\n", sizeof(v));
}

int main(void)
{
  float arreglo[TAM]; /* crea arreglo */

  printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));


  return 0;
}
```

Ejemplo: tamaño de un arreglo/puntero

```c
#include <stdio.h>
#define TAM 20

void imprimirTamanio(float v[])
{
  printf("Tamaño del arreglo: %lu\n", sizeof(v));
}

int main(void)
{
  float arreglo[TAM]; /* crea arreglo */

  printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));


  return 0;
}
```

```
Tamaño del arreglo: 80
```

Ejemplo: tamaño de un arreglo/puntero

```c
#include <stdio.h>
#define TAM 20

void imprimirTamanio(float v[])
{
  printf("Tamaño del arreglo: %lu\n", sizeof(v));
}

int main(void)
{
  float arreglo[TAM]; /* crea arreglo */

  printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
  imprimirTamanio(arreglo);

  return 0;
}
```

```
Tamaño del arreglo: 80
```

Ejemplo: tamaño de un arreglo/puntero

```c
#include <stdio.h>
#define TAM 20

void imprimirTamanio(float v[])
{
  printf("Tamaño del arreglo: %lu\n", sizeof(v));
}

int main(void)
{
  float arreglo[TAM]; /* crea arreglo */

  printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
  imprimirTamanio(arreglo);

  return 0;
}
```

```
Tamaño del arreglo: 80
Tamaño del arreglo: 8
```

Ejemplo: tamaño de un arreglo/puntero

```c
#include <stdio.h>
#define TAM 20

void imprimirTamanio(float v[TAM])
{
  printf("Tamaño del arreglo: %lu\n", sizeof(v));
}

int main(void)
{
  float arreglo[TAM]; /* crea arreglo */

  printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
  imprimirTamanio(arreglo);

  return 0;
}
```

```
Tamaño del arreglo: 80
Tamaño del arreglo: 8
```

Ejemplo: función para imprimir un arreglo

```c
#include <stdio.h>
#define TAM 20

void imprimirArreglo(int v[], int n)
{
  int i; /* subíndice */

  for(i = 0; i < n; i++)
      printf("%d ", v[i]);
  printf("\n");
}

int main(void)
{
  int v[TAM] = {1, 2, 3, 4, 5};

  imprimirArreglo(v, 10);
  return 0;
}
```

```
1 2 3 4 5 0 0 0 0 0
```

Ejemplo: función para imprimir un arreglo

```c
#include <stdio.h>
#define TAM 20

void imprimirArreglo(int v[], int n)
{
  int i; /* subíndice */

  for(i = 0; i < n; i++)
      printf("%d ", v[i]);
  printf("- Tamaño: %d\n", sizeof(v));
}

int main(void)
{
  int v[TAM] = {1, 2, 3, 4, 5};

  imprimirArreglo(v, 10);
  return 0;
}
```

```
1 2 3 4 5 0 0 0 0 0 - Tamaño: 8
```