

Programación en C

David Alejandro González Márquez

Clase disponible en: <https://github.com/fokerman/computingSystemsCourse>

El lenguaje C

Desarrollado por **Dennis Ritchie** en los Laboratorios Bell (1973).

Tiene tipos de datos **estáticos**, débilmente tipado.

Diseñado para la programación estructurada de **alto nivel**
pero con soporte para operar a **bajo nivel**.

El lenguaje C

Desarrollado por **Dennis Ritchie** en los Laboratorios Bell (1973).

Tiene tipos de datos **estáticos**, débilmente tipado.

Diseñado para la programación estructurada de **alto nivel**
pero con soporte para operar a **bajo nivel**.

Estandarización del lenguaje ANSI C, posteriormente
adoptado por ISO (C89, C90, C99, C11).

La adopción del estándar lo hace **portable** entre plataformas y arquitecturas.

Hola Mundo

```
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

Hola Mundo

```
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

#include <stdio.h>: Incluye una biblioteca de funciones para escribir en entrada/salida.

Hola Mundo

```
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

#include <stdio.h>: Incluye una biblioteca de funciones para escribir en entrada/salida.

int main(): Declara una función principal que va a ser nuestro programa.

Hola Mundo

```
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

#include <stdio.h>: Incluye una biblioteca de funciones para escribir en entrada/salida.

int main(): Declara una función principal que va a ser nuestro programa.

printf: Función para imprimir en pantalla con un formato específico.

Hola Mundo

```
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

#include <stdio.h>: Incluye una biblioteca de funciones para escribir en entrada/salida.

int main(): Declara una función principal que va a ser nuestro programa.

printf: Función para imprimir en pantalla con un formato específico.

return 0;: Termina la función principal y sale del programa con un cero.

Declaración de Variables

Las **variables** se declaran y se asignan mediante el operador igual (=).

Toda variable ocupa una determinada cantidad de bytes dada por su tipo.

Ejemplos:

```
int i;  
int pepe = 35;  
char unaLetra = 'a';  
char contador = 9;  
float numero = 5.3;
```

Sintaxis

tipo-de-variable	nombre-de-variable ;
------------------	----------------------

Tipo	Descripción
int	entero de 4 bytes
char	entero de 1 byte
float	punto flotante 32 bits
double	punto flotante 64 bits

Declaración de Variables

Las **variables** se declaran y se asignan mediante el operador igual (=).

Toda variable ocupa una determinada cantidad de bytes dada por su tipo.

Ejemplos:

```
int i;                                ← Variable entera de 4 bytes de nombre i  
int pepe = 35;  
char unaLetra = 'a';  
char contador = 9;  
float numero = 5.3;
```

Sintaxis

tipo-de-variable	nombre-de-variable ;
------------------	----------------------

Tipo	Descripción
int	entero de 4 bytes
char	entero de 1 byte
float	punto flotante 32 bits
double	punto flotante 64 bits

Declaración de Variables

Sintaxis

tipo-de-variable

nombre-de-variable ;

Las **variables** se declaran y se asignan mediante el operador igual (=).

Toda variable ocupa una determinada cantidad de bytes dada por su tipo.

Tipo	Descripción
int	entero de 4 bytes
char	entero de 1 byte
float	punto flotante 32 bits
double	punto flotante 64 bits

Ejemplos:

```
int i;           ← Variable entera de 4 bytes de nombre i
int pepe = 35;   ← Variable y asignación de un valor
char unaLetra = 'a';
char contador = 9;
float numero = 5.3;
```

Declaración de Variables

Sintaxis

tipo-de-variable

nombre-de-variable ;

Las **variables** se declaran y se asignan mediante el operador igual (=).

Toda variable ocupa una determinada cantidad de bytes dada por su tipo.

Tipo	Descripción
int	entero de 4 bytes
char	entero de 1 byte
float	punto flotante 32 bits
double	punto flotante 64 bits

Ejemplos:

```
int i;           ← Variable entera de 4 bytes de nombre i
int pepe = 35;   ← Variable y asignación de un valor
char unaLetra = 'a';   ← Variable de un byte y asignación de un carácter
char contador = 9;
float numero = 5.3;
```

Declaración de Variables

Sintaxis

tipo-de-variable

nombre-de-variable ;

Las **variables** se declaran y se asignan mediante el operador igual (=).

Toda variable ocupa una determinada cantidad de bytes dada por su tipo.

Tipo	Descripción
int	entero de 4 bytes
char	entero de 1 byte
float	punto flotante 32 bits
double	punto flotante 64 bits

Ejemplos:

```
int i;           ← Variable entera de 4 bytes de nombre i
int pepe = 35;   ← Variable y asignación de un valor
char unaLetra = 'a';   ← Variable de un byte y asignación de un carácter
char contador = 9;    ← Variable de un byte y asignación de un número
float numero = 5.3;
```

Declaración de Variables

Sintaxis

tipo-de-variable

nombre-de-variable ;

Las **variables** se declaran y se asignan mediante el operador igual (=).

Toda variable ocupa una determinada cantidad de bytes dada por su tipo.

Tipo	Descripción
int	entero de 4 bytes
char	entero de 1 byte
float	punto flotante 32 bits
double	punto flotante 64 bits

Ejemplos:

```
int i;           ← Variable entera de 4 bytes de nombre i
int pepe = 35;   ← Variable y asignación de un valor
char unaLetra = 'a';   ← Variable de un byte y asignación de un carácter
char contador = 9;    ← Variable de un byte y asignación de un número
float numero = 5.3;   ← Variable en punto flotante de 32 bits
```

Operadores

Las **expresiones** del lenguaje se construyen mediante **operadores** matemáticos y lógicos.

En C las expresiones resultan en valores numéricos.

El valor cero es interpretado como el valor lógico **falso**, mientras que un valor distinto de cero se interpreta como **verdadero**.

Operador	Descripción	Operador	Descripción
+	suma	<<	shift a izquierda
-	resta	>>	shift a derecha
*	multiplicación	!	not lógico
/	división	&&	and lógico
%	resto de división		or lógico
==	igual	~	not bit a bit
!=	distinto	&	and bit a bit
>	mayor		or bit a bit
>=	mayor o igual	^	xor bit a bit
<	menor		
<=	menor o igual		

Operadores

Las **expresiones** del lenguaje se construyen mediante **operadores** matemáticos y lógicos.

En C las expresiones resultan en valores numéricos.

El valor cero es interpretado como el valor lógico **falso**, mientras que un valor distinto de cero se interpreta como **verdadero**.

Operador	Descripción	Operador	Descripción
+	suma	<<	shift a izquierda
-	resta	>>	shift a derecha
*	multiplicación	!	not lógico
/	división	&&	and lógico
%	resto de división		or lógico
==	igual	~	not bit a bit
!=	distinto	&	and bit a bit
>	mayor		or bit a bit
>=	mayor o igual	^	xor bit a bit
<	menor		
<=	menor o igual		

Ejemplos:

```
int valor = (2 << 10) - (n + m) / 5;  
int cond = (b <= 6 || c > 3) && !(a == 5);  
int stat = (~0x111 | v) & 0x5555;
```

Declaración de Funciones

Las funciones pueden tener cualquier nombre, toman una lista de **parámetros** y solo pueden **retornar** un valor resultado.

Existe una función distinguida de nombre **main** que es la principal en un programa C.

Para llamar a una función se usa su nombre y se pasan los parámetros entre paréntesis. El resultado se puede asignar o no a una variable.

Sintaxis

```
tipo-de-retorno nombre-de-función ( parametros-de-la-función ) {  
    ... cuerpo-de-la-función... ;  
}
```

Ejemplos:

```
int suma(int a, int b) {      int main() {  
    int s = a + b;            int t;  
    return s;                t = suma(2,4);  
}                                return esPar(t);  
  
int esPar(int numero) {  
    return !(numero & 1);  
}
```

Estructuras de control (for)

La sentencia condición-comienzo se ejecuta una vez al comienzo del ciclo,

la condición-final se ejecuta en cada iteración hasta hacerse falsa,

y la sentencia de incremento se ejecuta al final de cada iteración.

Sintaxis

```
for ( [condición-comienzo] ; [condición-final] ; [incremento] ) {  
    ... cuerpo-del-ciclo ... ;  
}
```

Ejemplo:

```
int potenciaDe2(int n) {  
    int t = 1;  
    for(int i=0; i<n; i++) {  
        t = t * 2;  
    }  
    return t;  
}
```

Estructuras de control (while)

Mientras la sentencia condición se cumpla, es decir, sea verdadera, se ejecutará el cuerpo-del-while.

Dentro de cuerpo-del-while se deben alterar el contexto para que la condición sea falsa y se pueda terminar el ciclo.

Sintaxis

```
while ( [ condición ] ) {  
    ... cuerpo-del-while ... ;  
}
```

Ejemplo:

```
int potenciaDe2(int n) {  
    int t = 1;  
    int i = 0;  
    while( i != n ) {  
        t = t * 2;  
        i++;  
    }  
    return t;  
}
```

Estructuras de control (while)

Mientras la sentencia condición se cumpla, es decir, sea verdadera, se ejecutará el cuerpo-del-while.

Dentro de cuerpo-del-while se deben alterar el contexto para que la condición sea falsa y se pueda terminar el ciclo.

Sintaxis

```
while ( [condición] ) {  
    ... cuerpo-del-while ... ;  
}
```

Sintaxis do-while

```
do {  
    ... cuerpo-del-while ... ;  
} while ( [condición] );
```

Ejemplo:

```
int potenciaDe2(int n) {  
    int t = 1;  
    int i = 0;  
    while( i != n ) {  
        t = t * 2;  
        i++;  
    }  
    return t;  
}
```

Estructuras de control (if)

La sentencia condición determina qué bloque de código debe ser ejecutado.

bloque-verdadero se ejecuta si la condición es verdadera,

y bloque-falso se ejecuta si la condición es falsa.

Sintaxis

```
if( condición ) {  
    ... bloque-verdadero ... ;  
}  
else {  
    ... bloque-falso ... ;  
}
```

Ejemplo:

```
int collatz(int x) {  
    int res;  
    if( esPar(x) ) {  
        res = x / 2;  
    } else {  
        res = 3 * x + 1;  
    }  
    return res;  
}
```

Ejemplo

De los primeros n números enteros,
sumar los que no sean múltiplo de 5:

```
int sumNoMul5(int n) {  
    int acum = 0;  
    for(int i = 1; i <= n; i++) {  
        if(i % 5) {  
            acum = acum + i;  
        }  
    }  
    return acum;  
}
```

Sumar tres números de 4 bytes y saturar
el resultado a 1 byte con signo:

```
char sumaSaturada(int a, int b, int c) {  
    int suma = a + b + c;  
    if(suma > 127) { // 127d = 0x7F  
        suma = 127;  
    }  
    if(suma < -128) { // -128d = 0x80  
        suma = -128;  
    }  
    return suma;  
}
```

Declaración de estructuras de datos

Un struct funciona como un grupo de **datos ordenados**, o como un tipo de datos compuesto.

Cada dato dentro del struct tiene un **tipo** y un **orden**.

Los podemos declarar y usar como parámetros de funciones o como variables dentro de una función.

Sintaxis

```
struct [nombre-de-la-estructura] {  
    [tipo-de-variable-0] [nombre-de-variable-0];  
    ...  
    [tipo-de-variable-n] [nombre-de-variable-n];  
};
```

Ejemplo:

```
struct persona {  
    char* nombre;  
    int dni;  
    float altura;  
};  
  
struct vector {  
    double x;  
    double y;  
    double z;  
};  
  
struct persona p;  
p.nombre = "Pepe Lujan";  
p.dni = 32038084;  
p.altura = 1.68;  
  
struct vector v;  
v.x = 8.99;  
v.y = 2.5;  
v.z = 12.31;
```

Arreglos

Los arreglos consisten en una serie de datos del mismo tipo ubicados en memoria de forma **contigua**.

Tienen un tamaño dado por la **cantidad** de datos del arreglo.

Sintaxis

tipo-del-arreglo nombre-del-arreglo [tamaño-del-arreglo] ;

Ejemplo:

Arreglos

Los arreglos consisten en una serie de datos del mismo tipo ubicados en memoria de forma **contigua**.

Tienen un tamaño dado por la **cantidad** de datos del arreglo.

Sintaxis

tipo-del-arreglo nombre-del-arreglo [tamaño-del-arreglo] ;

Ejemplo:

int a[5];



Arreglos

Los arreglos consisten en una serie de datos del mismo tipo ubicados en memoria de forma **contigua**.

Tienen un tamaño dado por la **cantidad** de datos del arreglo.

Sintaxis

tipo-del-arreglo nombre-del-arreglo [tamaño-del-arreglo] ;

Ejemplo:

int a[5];

[0] [1] [2] [3] [4]
a [? | ? | ? | ? | ?]

Arreglos

Los arreglos consisten en una serie de datos del mismo tipo ubicados en memoria de forma **contigua**.

Tienen un tamaño dado por la **cantidad** de datos del arreglo.

Sintaxis

tipo-del-arreglo nombre-del-arreglo [tamaño-del-arreglo] ;

Ejemplo:

```
int a[5];  
a[0] = 42;
```

[0] [1] [2] [3] [4]
a [42] ? ? ? ?

Arreglos

Los arreglos consisten en una serie de datos del mismo tipo ubicados en memoria de forma **contigua**.

Tienen un tamaño dado por la **cantidad** de datos del arreglo.

Sintaxis

tipo-del-arreglo nombre-del-arreglo [tamaño-del-arreglo] ;

Ejemplo:

```
int a[5];  
a[0] = 42;  
a[1] = a[0] + 3;
```

[0] [1] [2] [3] [4]
a [42] [45] [?] [?] [?]

Arreglos

Los arreglos consisten en una serie de datos del mismo tipo ubicados en memoria de forma **contigua**.

Tienen un tamaño dado por la **cantidad** de datos del arreglo.

Sintaxis

tipo-del-arreglo nombre-del-arreglo [tamaño-del-arreglo] ;

Ejemplo:

```
int a[5];  
a[0] = 42;  
a[1] = a[0] + 3;  
a[3] = a[0] - a[1];
```

a [0] [1] [2] [3] [4]
42 45 ? -3 ?

Ejemplo

Calcular el módulo de un vector:

```
float modulo2(float x, float y) {  
    return sqrt( x * x + y * y);  
}
```

Estructura vector

```
struct vector {  
    float x;  
    float y;  
};
```

Calcular el promedio de los módulos de un arreglo de vectores:

```
float promedio(struct vector v[], int size) {  
    float acumulador = 0;  
    for(int i = 0; i < size; i++) {  
        float x = v[i].x;  
        float y = v[i].y;  
        acumulador = acumulador + modulo2(x,y);  
    }  
    return acumulador / (float)(size);  
}
```

Punteros

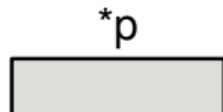
Los punteros son variables especiales que contienen la dirección de memoria donde se encuentra un dato específico.

Punteros

Los punteros son variables especiales que contienen la dirección de memoria donde se encuentra un dato específico.

Se **declaran** usando un * sobre el nombre de la variable:

```
char *p;
```



Punteros

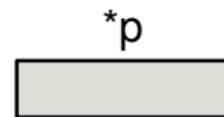
Los punteros son variables especiales que contienen la dirección de memoria donde se encuentra un dato específico.

Se **declaran** usando un * sobre el nombre de la variable:

```
char *p;
```

Se puede **obtener la dirección de memoria** de un dato usando un &:

```
&dato
```



Punteros

Los punteros son variables especiales que contienen la dirección de memoria donde se encuentra un dato específico.

Se **declaran** usando un * sobre el nombre de la variable:

```
char *p;
```

Se puede **obtener la dirección de memoria** de un dato usando un &:

```
&dato
```



Punteros

Los punteros son variables especiales que contienen la dirección de memoria donde se encuentra un dato específico.

Se **declaran** usando un * sobre el nombre de la variable:

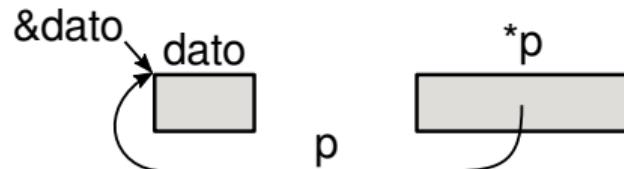
```
char *p;
```

Se puede **obtener la dirección de memoria** de un dato usando un &:

```
&dato
```

Se **accede** al dato usando un * sobre el nombre de la variable:

```
*p
```



Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

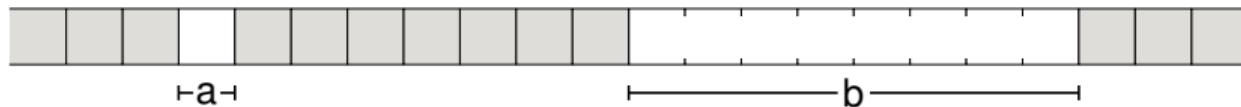
Ejemplos:

Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;
```

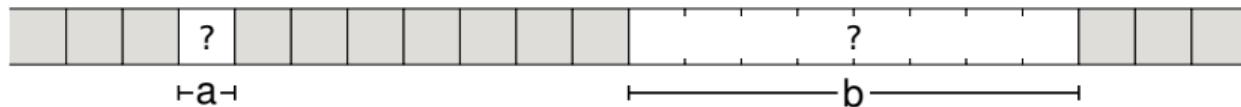


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;
```



Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;
```

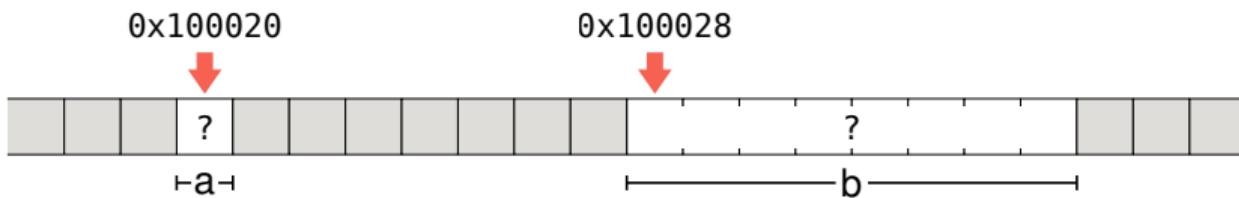


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;
```

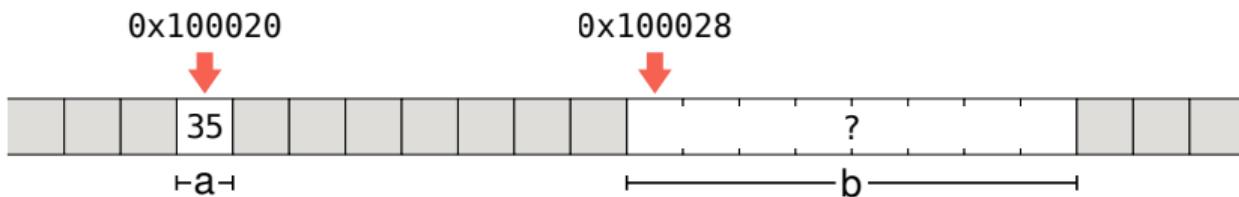


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;
```

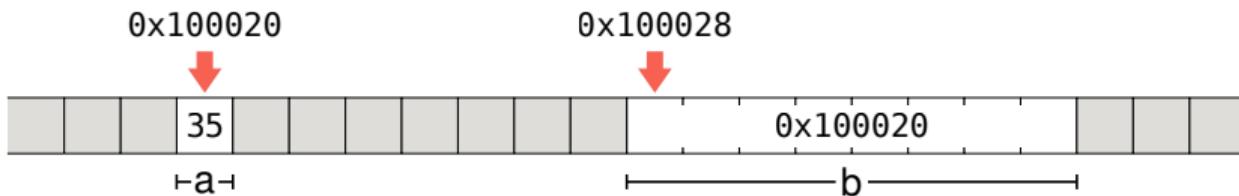


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;
```

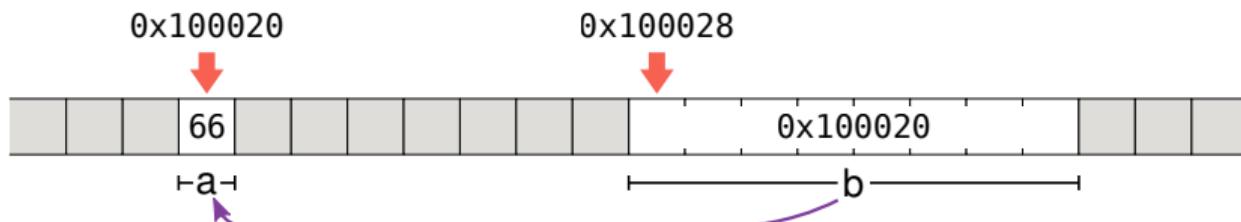


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```

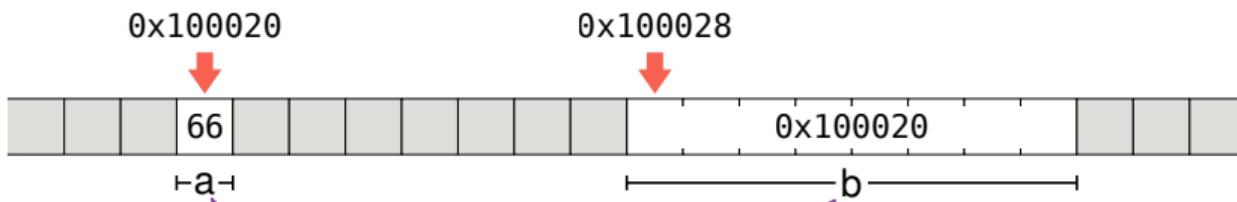


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;
```

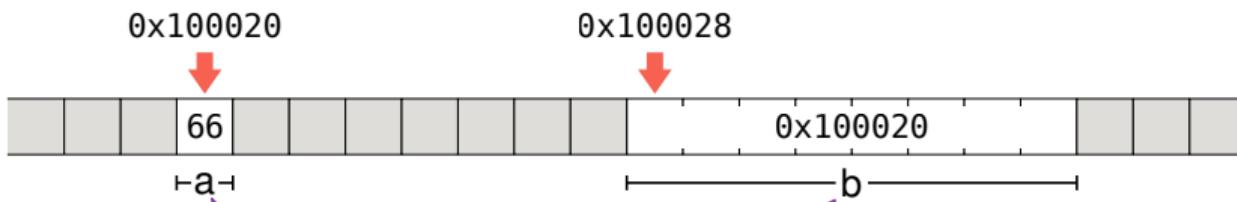


Punteros

El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;
```

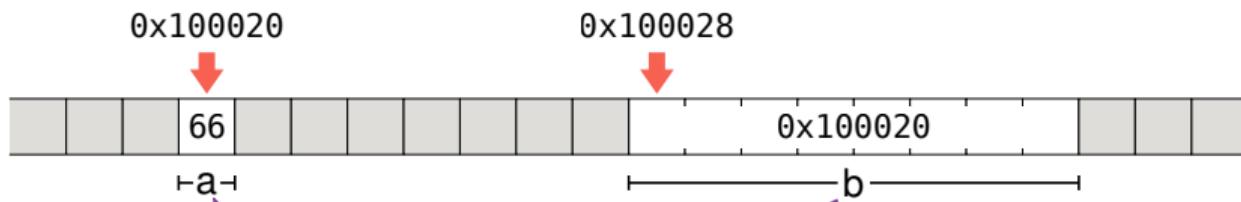


Punteros

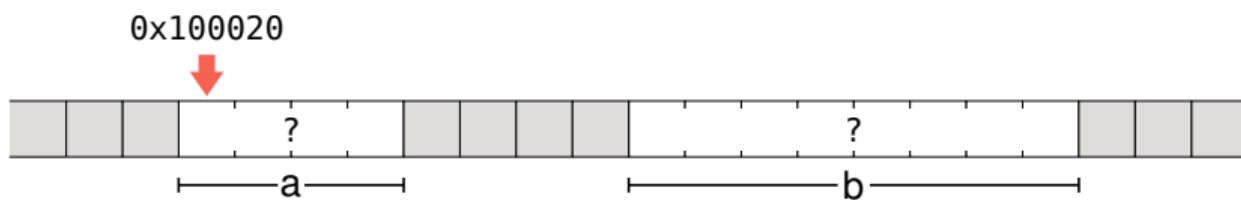
El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;
```

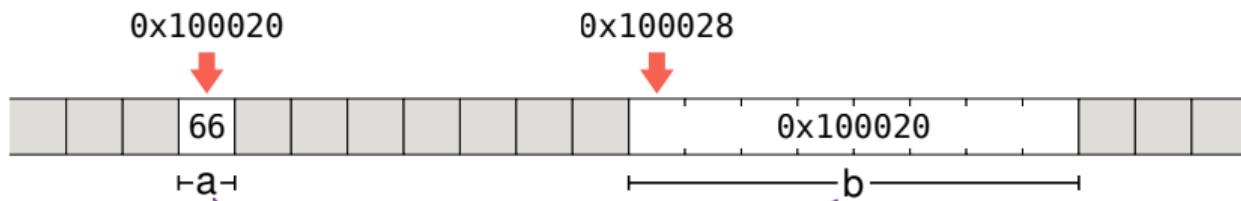


Punteros

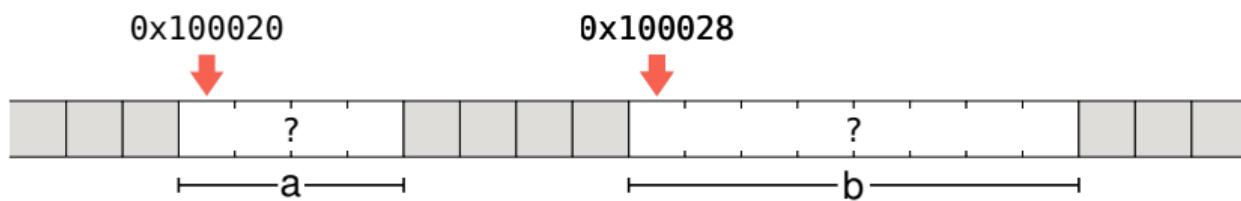
El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;
```

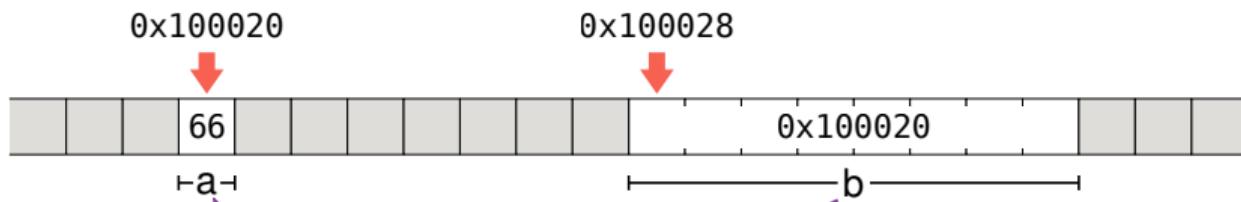


Punteros

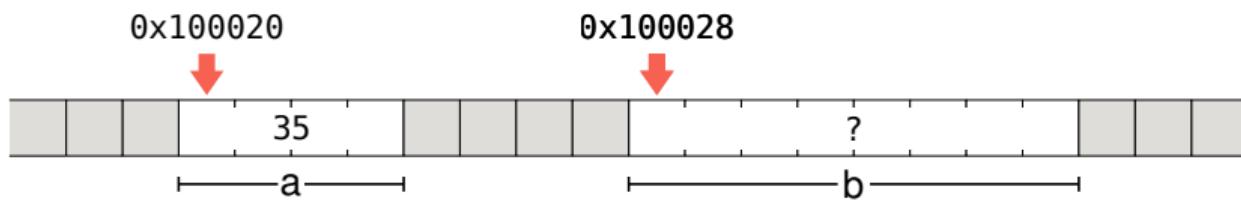
El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;  
a = 35;
```

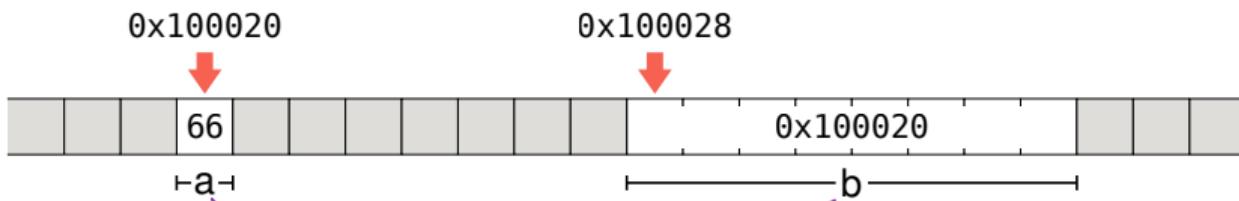


Punteros

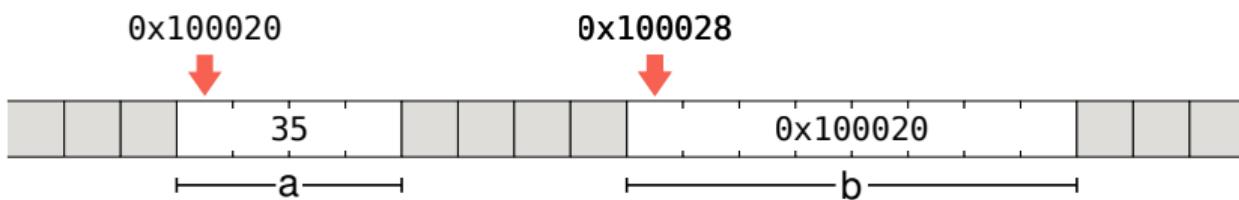
El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;  
a = 35;  
b = &a;
```

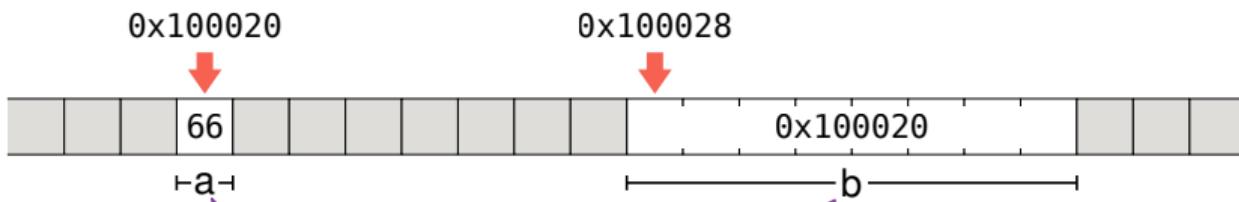


Punteros

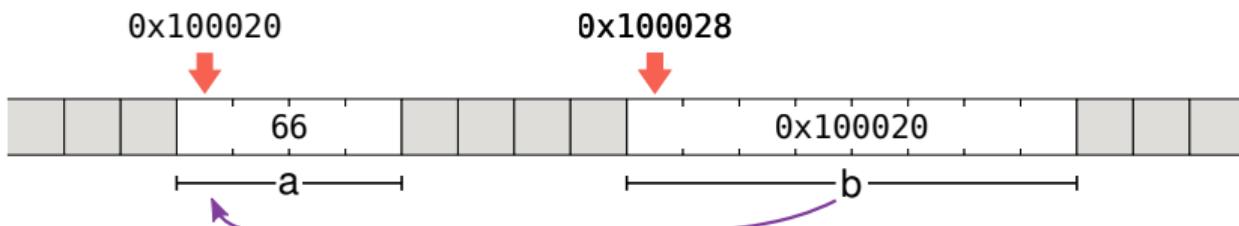
El tamaño en memoria de un puntero depende del direccionamiento a memoria.
En una arquitectura de 64 bits con direcciones de 64 bits, los punteros tienen 8 bytes.

Ejemplos:

```
char a;  
char *b;  
a = 35;  
b = &a;  
*b = 66;
```



```
int a;  
int *b;  
a = 35;  
b = &a;  
*b = 66;
```



Ejemplo

Función que suma usando solo punteros:

```
void suma(int* a, int* b, int* c) {  
    *c = *a + *b;  
}
```

Sumar los primeros n números enteros usando la función anterior:

```
int sumaEnteros(int n) {  
    int sumatoria = 0;  
    for(int i = 0; i < n; i++) {  
        suma(&sumatoria, &i, &sumatoria);  
    }  
    return sumatoria;  
}
```

Ejemplo

Calcular el módulo de un vector:

```
float modulo2(struct vector* v) {
    return sqrt( v->x * v->x + v->y * v->y);
}
```

Estructura vector

```
struct vector {
    float x;
    float y;
};
```

Calcular el promedio de los módulos de un arreglo de vectores:

```
float promedio(struct vector* v, int size) {
    float acumulador = 0;
    for(int i = 0; i < size; i++) {
        acumulador = acumulador + modulo2(&v[i]);
    }
    return acumulador / (float)(size);
}
```

Bibliografía

- Brian Kernighan y Dennis Ritchie. "El lenguaje de programación C", 2da Edición, 1991.
- Mike Banahan, Declan Brady and Mark Doran, "The C Book", 2da Edición, 1991.
Disponible online: https://publications.gbdirect.co.uk/c_book/

¡Gracias!

Recuerden leer los comentarios adjuntos
en cada clase por aclaraciones.