

TECNOLOGICO
NACIONAL DE MEXICO

U1- Programación Orientada a objetos PUNTEROS (Pointers)

Arquitectura de Programación para el control de Hardware

TECNOLOGICO
NACIONAL DE MEXICO

ARQ. DE PROG. PARA HARDWARE

INTRODUCCIÓN

Ingeniería
Electrónica

¿Punteros?

- Los punteros son un tipo especial de datos **C++** cuyo fin específico es almacenar direcciones de objetos.
- Comparten las características de las variables. Es decir: tienen un valor (tienen Rvalue); pueden ser asignados (tienen Lvalue);
- tienen un álgebra específica (se pueden realizar con ellos determinadas operaciones);
 - pueden ser almacenados en matrices;
 - pasados como parámetros a funciones y devueltos por estas.
- Comprenden dos categorías principales: **punteros-a-objeto** y **punteros-a-función** según el tipo de objeto señalado. Ambas categorías comparten ciertas operaciones, pero tienen uso, propiedades y reglas de manipulación distintas.

Punteros

- Variables que almacenan direcciones de otras variables.
- En general decimos coloquialmente que un puntero "apunta" o "señala" a un objeto determinado cuando su valor (del puntero) es la dirección del objeto (dirección de memoria donde comienza su almacenamiento).
- El objeto señalado por el puntero se denomina **referente**. Por esta razón también se dice que el puntero "referencia" al objeto.
- El operador que permite obtener la dirección de un objeto (para asignarlo a un puntero) se denomina operador de "**referencia**" (&)

Punteros

- La operación de obtener el referente a partir del puntero se denomina "**dereferenciar**" con el **Operador de indirección *** o de "**deferencia**", este devuelve el valor del objeto señalado por el operando.
- **Nota:** recordemos que el símbolo * tiene tres usos en C++: **operador de multiplicación** ([4.9.1](#)), indicador de tipo de variable (tipo puntero [4.2.1a](#)) y **operador de indirección**, que es el que aquí nos ocupa.

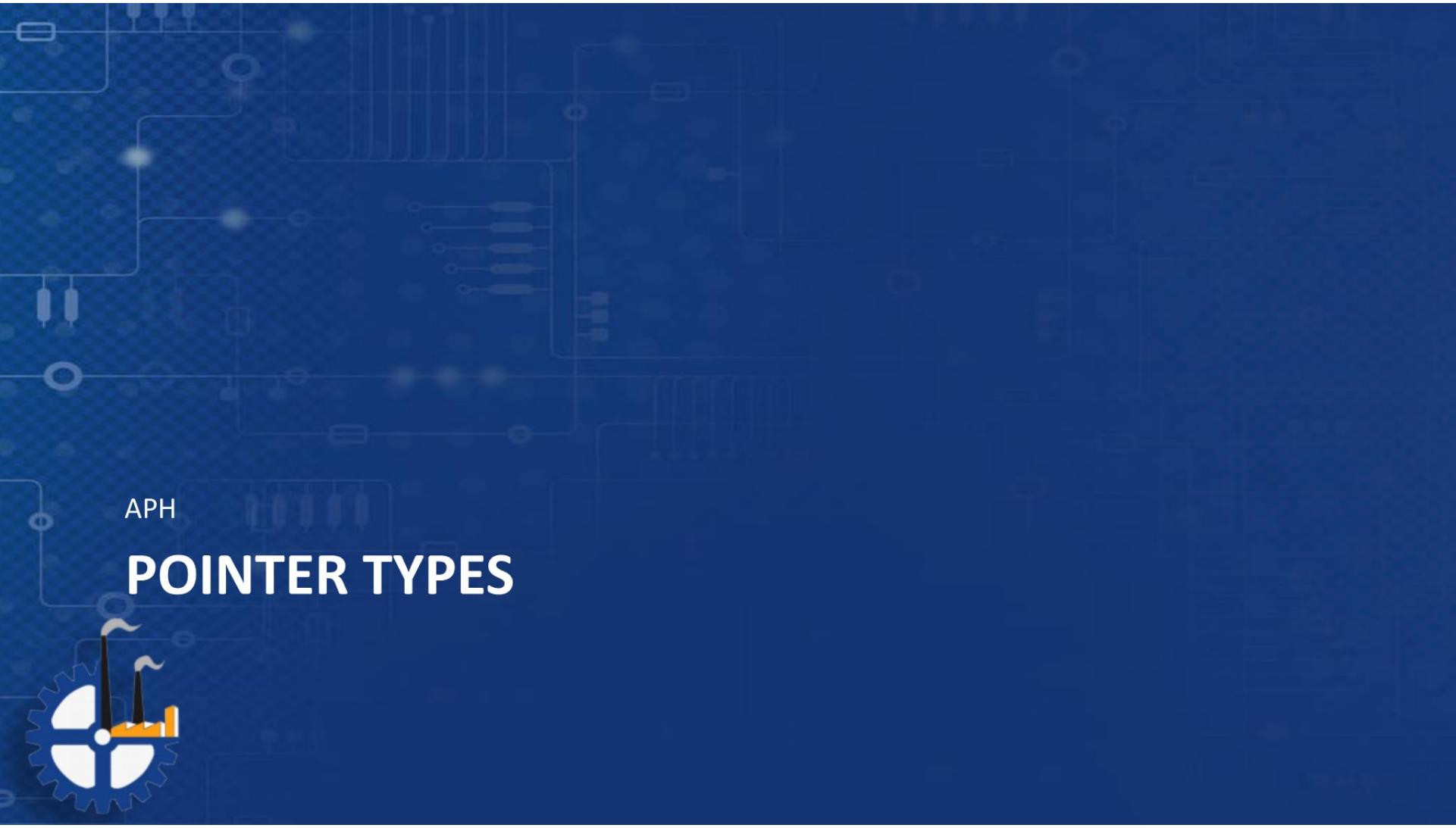
Punteros

Nota: tradicionalmente se considera que punteros y matrices están estrechamente relacionados [1], hasta el punto de ser estudiados simultáneamente.

De hecho, cualquier operación que puede efectuarse entre subíndices de matrices puede efectuarse con punteros, en general de forma más rápida (aunque algo más difícil de entender por el principiante).

Además, el operador de elemento de matriz [] se define en términos de ser un puntero ([4.9.16](#)).

-

A dark blue background featuring a faint, glowing circuit board pattern with various resistors, capacitors, and inductors.

APH

POINTER TYPES



Tipos de Puntero

Tipo de puntero denota el **tipo de dato** que el puntero va a *dereferenciar*



Tipo de dato
a
derefenciar

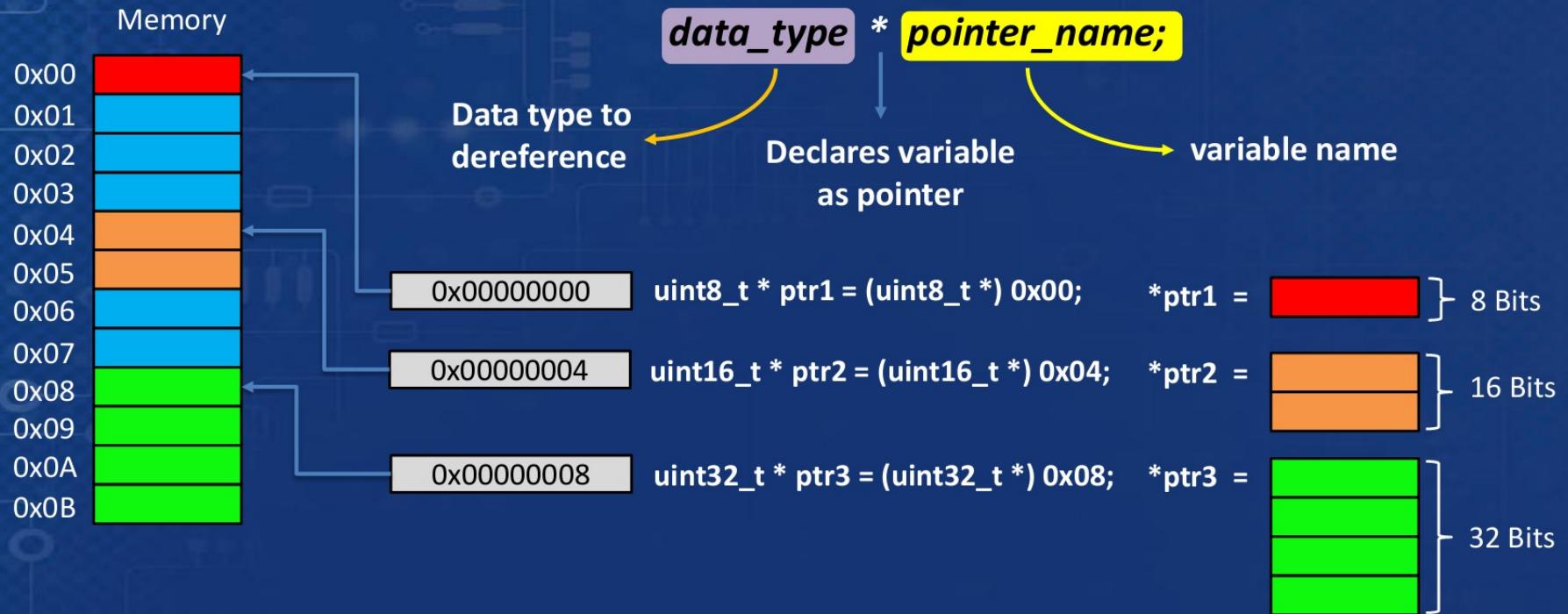
`data_type * pointer_name;`

Declara la variable
como puntero

Nombre de la
variable

Tipos de Puntero

Tipo de puntero denota el **tipo de dato** que el puntero va a **dereferenciar**



Tipos de Puntero

- Al tratar con punteros es importante tener muy claro que los punteros señalando a objetos de tipo distinto, son objetos de tipo distinto entre sí. Por ejemplo:

```
int *pt1;          // puntero-a-entero
int const *pt2;   // puntero-a-entero constante
```

- En este caso pt1 y pt2 son punteros de distinto tipo (aunque ambos señalen a valores de tipo **int**) y por tanto **no** pueden realizarse asignaciones directas entre ellos. Por ejemplo:

```
int const peso = 50; // declara peso constante tipo int
int const *pt1;    // declara pt1 puntero-a-int-constante
pt1 = &peso;       // Ok asignación permitida.
int *pt2;          // declara pt2 puntero-a-int
pt2 = &peso;        // Error: el puntero no es adecuado
```

Tipos de Puntero

- Una vez declarado, un puntero permanece encasillado; no puede señalar a objetos de tipo distinto que el original. La excepción, con ciertas limitaciones (Punteros genéricos) son los punteros declarados inicialmente como puntero-a-void. Por esta razón, este tipo de punteros son considerados **punteros genéricos** (no debe ser confundido con **puntero nulo**).
- Ejemplo:

```
int x = 20;           // declara x tipo int
void* pt3;           // declara pt3 puntero-a-void (genérico)
pt3 = &x;            // Ok asignación permitida
```

Tipos de Puntero

- No obstante el "encasillado" antes aludido, el **modelado de tipos** puede utilizarse también con punteros, permitiendo reasignarlos. Por ejemplo, siguiendo con el caso anterior, sería posible hacer:

```
int* pt4;           // declara pt4 puntero-a-int
pt4 = (int*) &peso; // Ok: asignación permitida
```

- Al realizar operaciones con punteros modelados, es posible que se produzcan corrupciones de memoria o machacamiento de datos, ya que la aritmética de punteros tiene en cuenta el tamaño del objeto señalado.
- La asignación entre un puntero-a-**tipo1** y un puntero-a-**tipo2** sin un moldeado apropiado, puede producir un mensaje de aviso, o error del compilador, cuando **tipo1** y **tipo2** son distintos. Si **tipo1** es una función y **tipo2** no, o viceversa, la asignación será siempre ilegal.

A dark blue background featuring a faint, glowing circuit board pattern with various resistors, capacitors, and inductors.

APH

POINTER SIZE

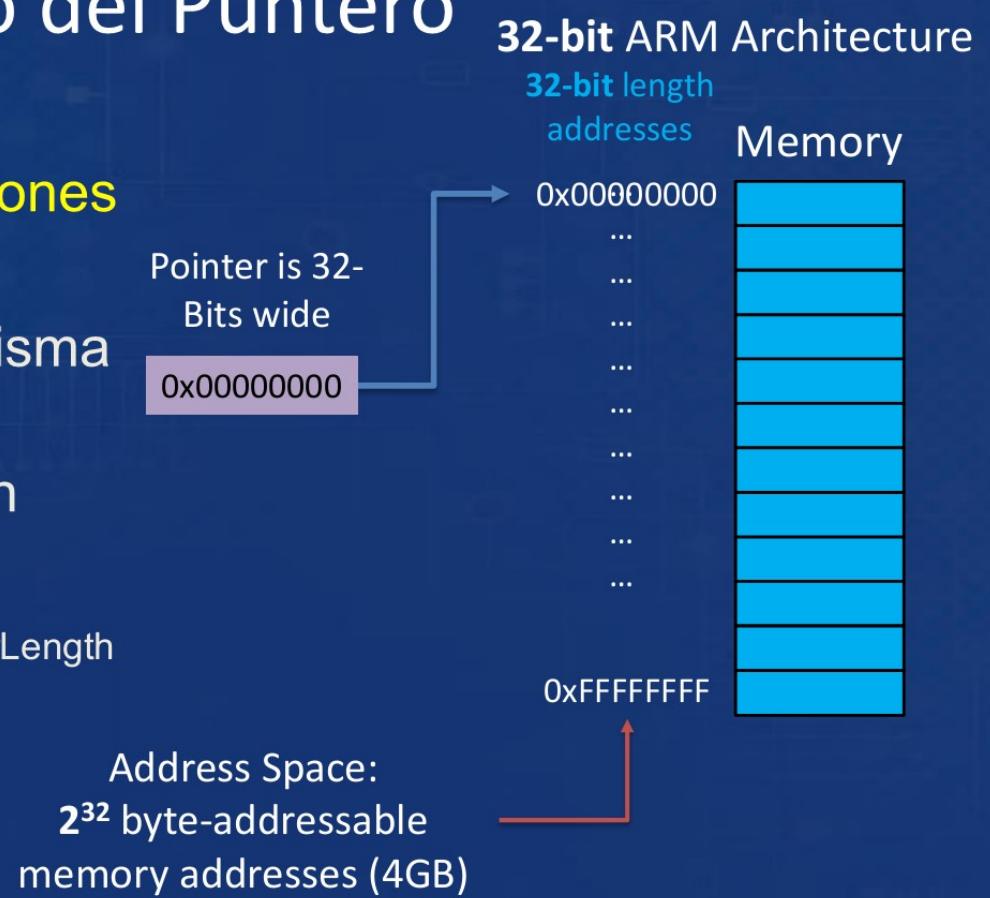


Tamaños de Punteros

- Su **tamaño** es el suficiente para almacenar una dirección de memoria en la arquitectura de la computadora utilizada.
- En la mayoría de compiladores de **32** bits los punteros ocupan **4 bytes**, es decir, coincide con el de un entero, de forma que **sizeof(int) == sizeof(void*)** no obstante, no tiene porqué ser así necesariamente.
- Por ejemplo, en algunos compiladores para 64 bits, el tamaño del puntero es mayor que el tamaño de un entero.

Tamaño del Puntero

- Los punteros mantienen **direcciones** a ubicación en la memoria
- Todos los punteros son de la misma longitud
 - ARM → 32-bit Pointer Length
- Espacio de Direcciones= $2^{\text{PointerLength}}$



Tamaño del Puntero

- Todos los punteros son de la misma longitud

```
uint8_t * ptr1 = (uint8_t *) 0x00;  
uint16_t * ptr2 = (uint16_t *) 0x04;  
uint32_t * ptr3 = (uint32_t *) 0x08;  
float * ptr4 = (float *) 0x0C;
```

`sizeof(uint8_t*)` = `sizeof(int16_t*)`
= `sizeof(uint32_t*)`
= `sizeof(float*)`
= 32-Bits!

`sizeof(ptr1)` = `sizeof(ptr2)`
= `sizeof(ptr3)`
= `sizeof(ptr4)`
= 32-Bits!

- Los punteros Dereferencian datos de diferente tamaño

`sizeof(*ptr1)` ≠ `sizeof(*ptr2)` ≠ `sizeof(*ptr3)` ≠ `sizeof(*ptr4)`

1 Byte	2 Byte	4 Byte	4 Byte
--------	--------	--------	--------

APH

POINTER OPERATORS

Operadores de Punteros

- Operador de dereferencia= *****
 - Accesses data at address
- Operador de Dirección= **&**
 - Provides address of variable
- Enteros no son direcciones
- Cast a dirección explícita para Memoria periférica

```
uint32_t var;
```

```
uint32_t * ptr = &var;
```

```
*ptr = 0xABCD1234;
```

```
uint16_t * ptr = (uint16_t *) 0x480C0000;
```



Casts Integer to address

¿Typecasting?

- El **modelado de tipos** ("typecasting") es el proceso de convertir o promover un objeto de un tipo a otro.
- Esta operación es necesaria y frecuente en programación; incluso es realizada infinidad de veces por el compilador de forma automática y transparente para el programador, y ocurre cuando este la solicita de **forma implícita**.
 - Por ejemplo, cuando una función en cuyo prototipo se ha declarado que espera un **float** como argumento y le pasamos un entero, o cuando necesitamos sumar un **int** con un **double**.
- En todos estos casos, el compilador realiza automáticamente una conversión de tipo para que el argumento pasado concuerde con el esperado. En el caso de operaciones aritméticas, la conversión intenta conseguir la menor pérdida de precisión posible en las operaciones.

Casting

- Es el estilo heredado de C que se mantiene por compatibilidad, tiene dos variedades de notación.

Sintaxis:

(<Nombre-de-tipo>) <expresión>
<Nombre-de-tipo> (<expresión>)

El valor de <expresión> se convierte al tipo definido por <nombre-de-tipo> siguiendo las reglas estándar de conversión.

Ejemplo

- Por ejemplo, la función de la Librería Estándar **sqrt()** espera un **double** como argumento, si queremos utilizarla con un **int n**, se puede utilizar cualquiera de las expresiones siguientes:

```
sqrt( (double) n );      // primera forma de la sintaxis  
sqrt( double (n) );    // segunda forma de la sintaxis
```

Observe que n no se altera, ya que la conversión se realiza antes de pasar el argumento a la función.

Observe también que si los parámetros se han especificado en el prototipo de la función, esta conversión de argumento no sería necesaria, ya que el compilador realiza automáticamente el modelado correspondiente.

Ejemplo

- Las expresiones que siguen son válidas en C++:

```
double peso = 25;
int * ptr;
ptr = (int *)&peso;
cout << (long)&peso;      // para que se muestre en decimal
return (int) peso;
func (double doble) { return (int) doble+3; }
```

- En ocasiones se puede invertir la colocación del paréntesis (segundo caso de la sintaxis), utilizando entonces una notación parecida a las funciones:

```
return int (peso);
func (double doble) { return int (doble+3); }
```

APH

NULL POINTER E INICIALIZACIÓN



Null Pointer

- Un **puntero nulo** es un puntero que apunta a una dirección que no corresponde a ningún objeto del programa. Asignando la constante entera 0 a un puntero se le asigna el valor nulo. Para mejor legibilidad puede utilizarse la constante manifiesta **NULL** (definida en la librería de cabeceras estándar **<stdio.h>**). Ejemplo:

```
int* punt1;           // L.1: Declara punt1
punt1 = 0;            // L.2: define punt1 como puntero nulo
punt1 = NULL;         // Equivalente a la anterior
```

- **Nota:** la cuestión de la inicialización del puntero nulo es uno de los puntos de controversia para los teóricos del lenguaje. En rigor, la sentencia L.2 anterior es errónea, ya que el tipo de punt1 es puntero-a-int, mientras que 0 es una constante numérica (**int** por defecto); a pesar de lo cual es aceptada sin rechistar por el compilador . Lo sintácticamente correcto sería realizar un modelado adecuado :

```
punt1 = static_cast<int*>(0);
```

Punteros nulos

- En el momento de la declaración del puntero, es posible que no conozca la dirección

→ Use a NULL Pointer for Initialization

- Null Pointers apuntan a nada.
 - Se usa para verificar si hay un puntero válido
- Desreferenciar un puntero NULL puede causar una excepción

This pointer will have garbage data

```
uint32_t * ptr;
```

Using un-initialized pointer can potentially corrupt your memory

```
#define NULL ((void*)0)  
uint32_t * ptr = NULL;
```

```
if (ptr == NULL) {  
    /* error! */  
}  
*ptr = 0xABCD1234;
```

Incialización de Punteros

- En general los punteros pueden ser inicializados como otra variable cualquiera; desde luego solo tiene sentido que el valor sea la dirección de inicio del almacenamiento de un objeto del tipo adecuado. En otras palabras: un valor que sea una dirección de memoria.
- La forma usual de inicializar un puntero suele ser asignándole el valor devuelto por dicho operador. Ejemplo:

```
int x = 100;
int* ptr;           // declaración de ptr como puntero-a-int
ptr = &x;           // se le asigna la "dirección" de una variable
```

Incialización de Punteros

- También es muy frecuente iniciarlos con el valor devuelto por el operador **new** que precisamente devuelve un puntero a objeto. Ejemplo:

```
class C { .... };      // definición de una clase C
C* cptr;              // declara cptr como puntero-a-C
cptr = new C(...);    // new crea una instancia de la clase y devuelve un
                     // puntero al objeto, cuyo valor que es asignado a cptr
```

- **Nota:** los enteros y los punteros **no** son intercambiables, con excepción del cero. El cero es un valor que nunca puede adoptar un puntero en condiciones normales, por esta razón se utiliza para significar "no asignado" u otra circunstancia especial. Recordar que la aritmética de punteros permite comparar dos punteros, para igualdad o desigualdad, con **NULL**.

Punteros como argumentos

- En muchas ocasiones los punteros se utilizan como argumentos a funciones. De hecho, la mayoría de las veces en que las matrices o las cadenas de caracteres pasan como argumentos, lo que en realidad pasa es un puntero al primer elemento. Por ejemplo:

```
char* str = "La Tacita de Plata";
printf("La ciudad es: \"%s\"\n", str);
```

- En este caso, la función **printf** recibe como segundo argumento str, que es un puntero a la posición de memoria donde comienza el almacenamiento de la cadena "La Tacita de Plata\0".

Punteros como argumentos

- En cualquier caso, es necesario declarar esta circunstancia en el prototipo y en la definición de la función para que el compilador conozca que el argumento pasado es una variable tipo puntero.

Ejemplo:

```
# include <iostream.h>

void func(int *);      // prototipo: argumento definido como puntero-a-int

void main () {          // =====
    int x = 35;
    int * ptr = &x;
    func (ptr);           // se pasa puntero-a-int como argumento
    cout << "El valor es ahora: " << x << endl;
}

void func (int * p) { // el argumento se define como puntero-a-int
    *p += 10;           // el argumento es tratado como puntero
}
```

- Salida:

El valor es ahora: 45

Ejemplo de Puntero (con struct)

```
typedef struct foo {  
    uint8_t varA;  
    uint8_t varB;  
    uint16_t varC;  
} foo_t;
```

```
foo_t varS;  
uint8_t Num;
```

```
foo_t * varS_ptr = &varS;  
uint8_t * ptr_Num = &Num;
```

At least
32 Bits

Each Pointer is 32-bits

Ejemplo de Puntero (con struct)

```
typedef struct foo {  
    uint8_t varA;  
    uint8_t varB;  
    uint16_t varC;  
} foo_t;
```

```
foo_t varS;  
uint8_t Num;
```

```
foo_t * varS_ptr = &varS;  
uint8_t * ptr_Num = &Num;
```

varS_ptr->varA → Dereferences 8-bits
varS_ptr->varB → Dereferences 8-bits
varS_ptr->varC → Dereferences 16-bits

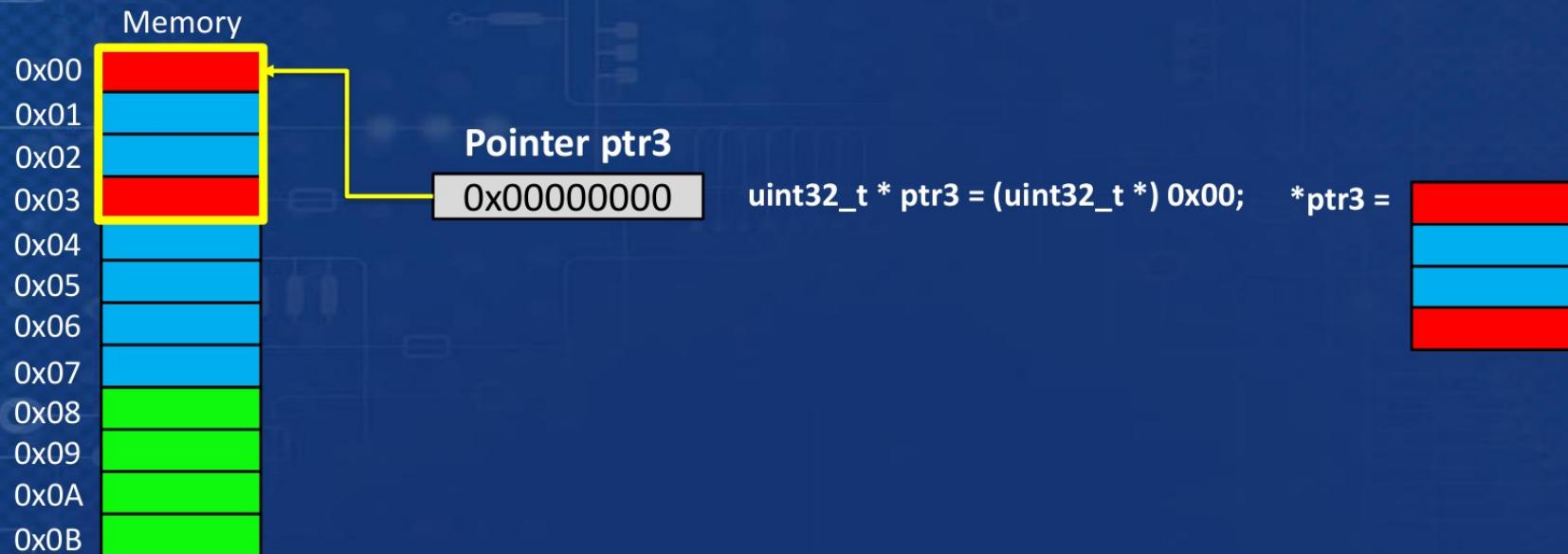
*ptr_Num → Dereferences 8-bits

varS_ptr->varC

Structure Pointer Dereference Operator

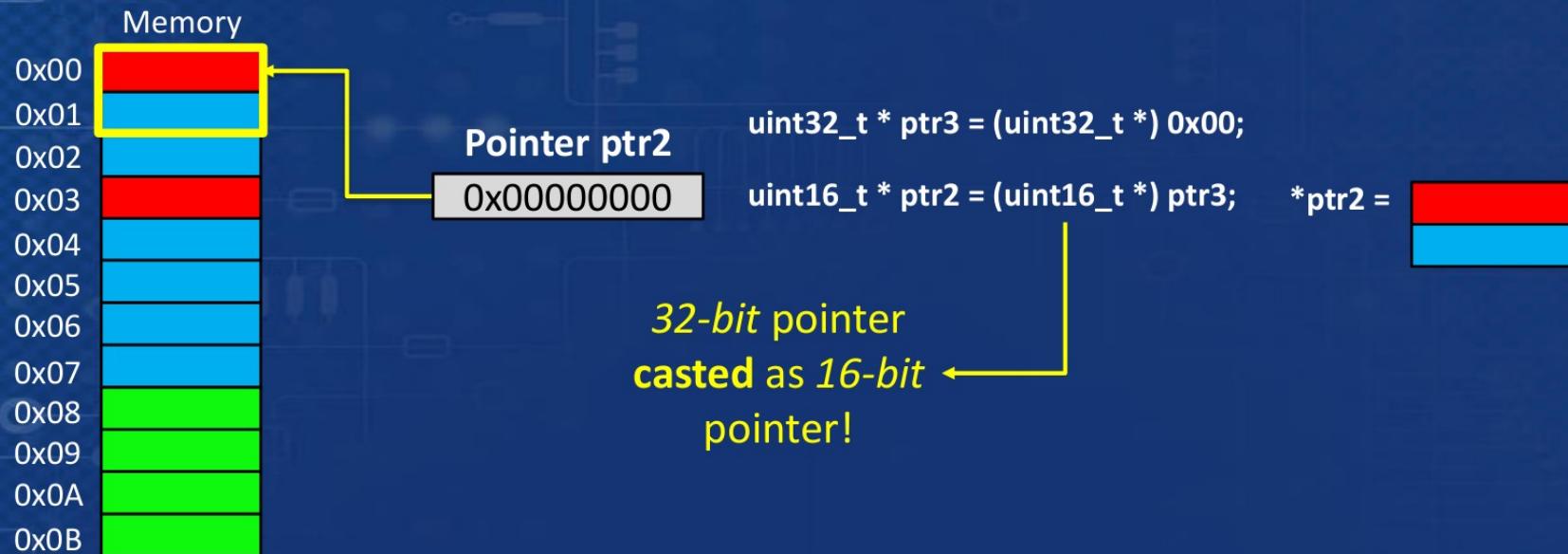
Pointer Casting

Cast de punteros para dereferenciar diferentes tamaños de la misma dirección



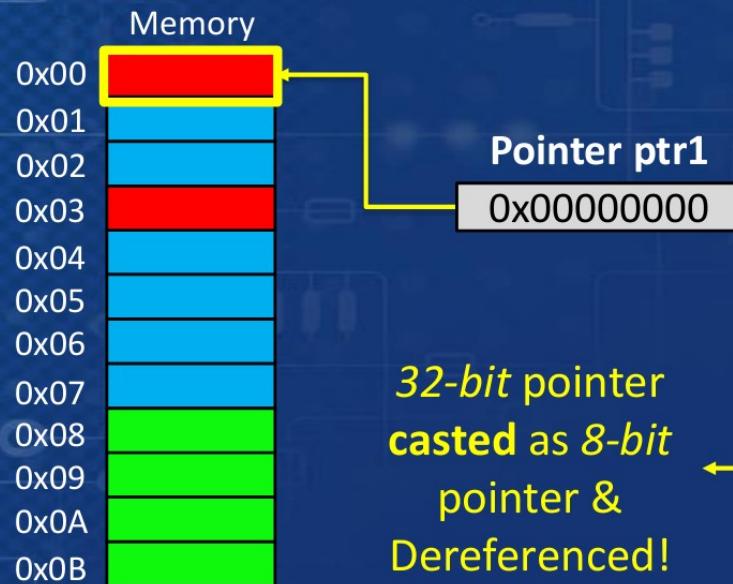
Pointer Casting

Cast de punteros para dereferenciar diferentes tamaños de la misma dirección



Pointer Casting

Cast de punteros para dereferenciar diferentes tamaños de la misma dirección



```
uint32_t * ptr3 = (uint32_t *) 0x00;
```

```
uint16_t * ptr3 = (uint16_t *) ptr3;
```

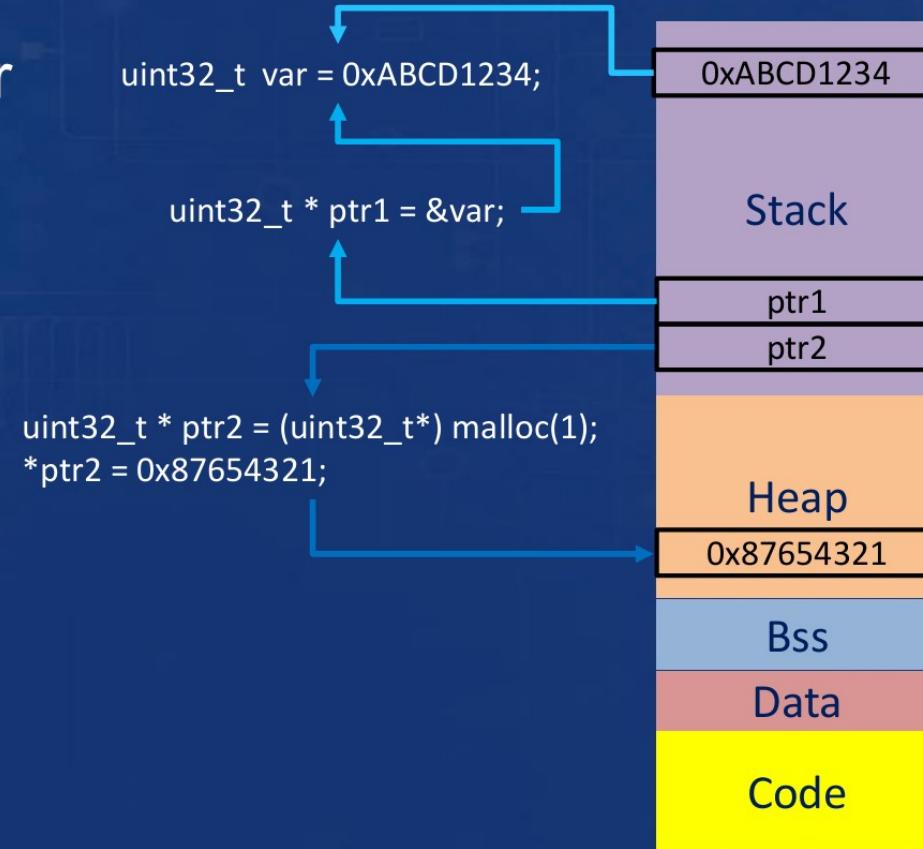
```
uint8_t * ptr1 = (uint8_t *) ptr3;
```

```
*ptr1 =
```

```
*((uint8_t*)ptr3) =
```

Punteros en Memoria

- Punteros existen en cualquier parte de la Memoria
 - Stack, Heap, BSS, Data
- Punteros pueden referenciar datos en diferentes partes de la memoria
 - Code, Data, Peripheral





¡Gracias!