



# INF 263 – Algoritmia

## Punteros

---

MG. JOHAN BALDEON

MG. RONY CUEVA MOSCOSO

2021-1

A solid orange horizontal bar spans the width of the slide at the bottom.

# Contenido

---

## 1. Arreglos

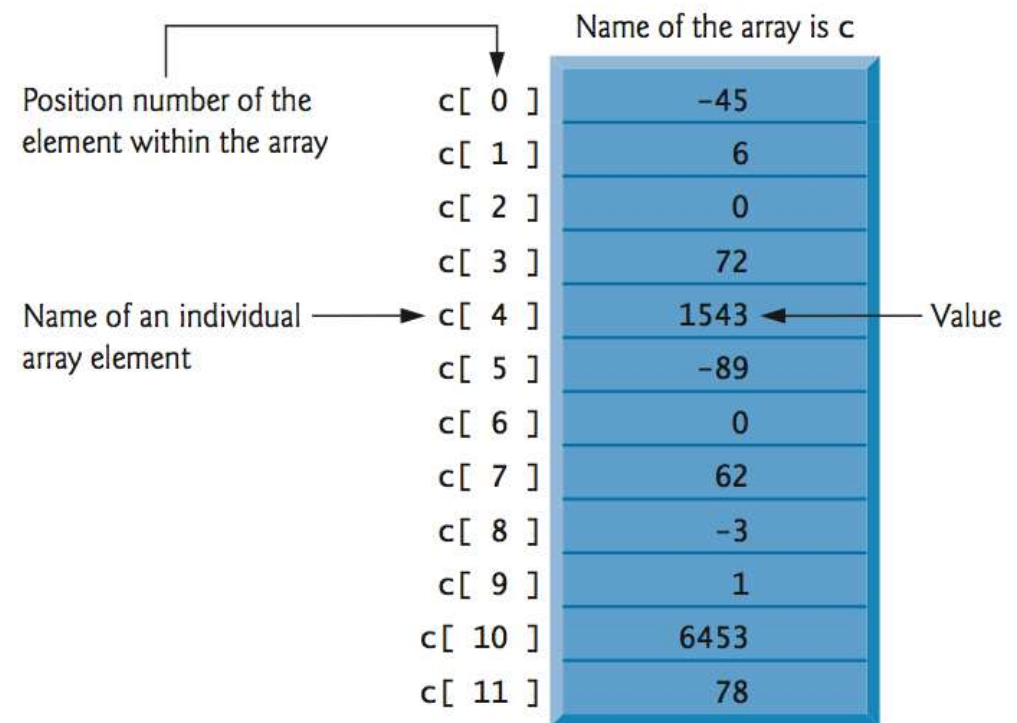
- Declaración
- Inicialización
- Arreglos como parámetros

## 2. Punteros

- Declaración
- Operadores (dirección / indirección)
- Parámetros por referencia
- Aritmética de punteros
- Punteros con memoria dinámica

# Arreglos (*Built-in Array*)

- Colección de espacios de memoria consecutivos para un **mismo tipo de dato**
- Estructuras de datos **estáticas** (su tamaño no varía)
- Operador de acceso: `[]`



# Arreglos: Declaración

---

**<tipo>** **<nombre\_arreglo>** [**tamaño\_arreglo**]

**int** a[10];

**char** c[30];

**const int size** = 10;  
**int** a[size];

También se puede declarar con una expresión **constante** (**const**) que no puede modificarse luego

# Arreglos: Iniciación

---

```
int a[10];  
for (int i=0; i<10; i++) a[i]=0;
```

```
int a[10] = {4, 67, 2, -1, 0,  
            89, 10, 0, -34, 1};
```

```
int a[10] = {4, 67, 2, -1};
```

```
int a[10] = {0};
```

Si el número de  
**inicializadores** es  
**menor** al **tamaño**  
entonces se completan  
los valores con **cero (0)**

# Arreglos: Paso como parámetro

---

El valor del “**nombre del arreglo**” (identificador) es la **dirección de memoria** del **primer elemento** del arreglo:

```
int a[10];
```

Asimismo, se “sugiere” enviar también el **tamaño** del arreglo:

```
foo(a, 10);
```

# Arreglos: Paso como parámetro

---

El compilador, internamente, no distingue lo siguiente:

```
void foo(int values[], int size){...};
```

```
void foo(int * values, int size){...};
```

Si no se quiere que una función modifique el contenido del arreglo, se puede usar el calificador **const**

```
void foo(const int values[], int size){...};
```

```
void foo(const int * values, int size){...};
```

# Punteros

---

*“Pointers:  
one of the **most powerfull**, yet **challenging** to use,  
**C++ capabilities**”  
(Deitel, 2015)*

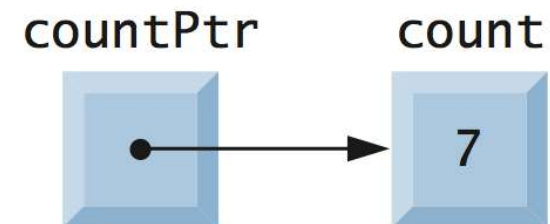
*“With great **power**, comes great **responsability**”  
Uncle Ben*



# Punteros

---

- Es un **tipo de variable** cuyo valor corresponde a una **dirección de memoria**
- Un puntero **referencia indirectamente a un valor** (el que está guardado en la dirección de memoria)
- Son útiles para el **paso como referencia** y en el uso de **memoria dinámica**

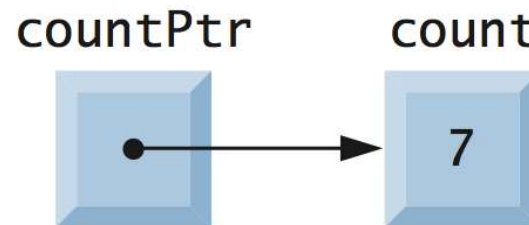


# Punteros: Declaración

---

**<tipo\_var\_apuntada>** \* **<nombre\_puntero>**

**int** \* countPtr;



Un puntero se “vincula” a un **tipo de dato** desde su declaración

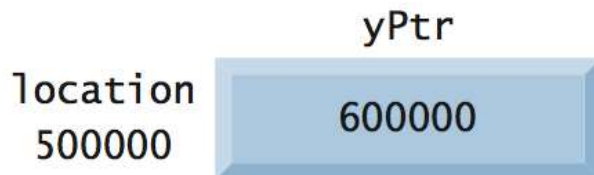
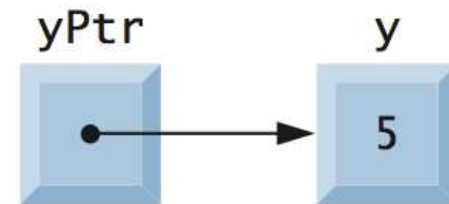
# Punteros:

## Operador de Dirección (&)

Permite obtener la **dirección de memoria** del **operando**:

Asigna la dirección de memoria de **y** a **yPtr**

```
int y = 5;  
int *yPtr;  
yPtr = &y;
```



# Punteros:

## Operador de Indirección (\*)

---

Retorna un “**alias**” del elemento al cual se dirige la variable puntero.

Permite **manipular el valor** de la dirección de memoria apuntada:

```
*yPtr = 9;
```

```
//Imprime el valor apuntado
```

```
printf("%d", *yPtr);
```

```
//Recibe datos de entrada
```

```
scanf("%d", yPtr);
```

# Punteros: Parámetros por Referencia

---

Permiten **pasar por referencia** los parámetros de funciones:

- Permite modificar una o más variables en la función invocada
- Permite enviar grandes objetos de datos sin crear “copias”

```
int main() {  
    int a=5;  
    foo( & a );  
    return 0; }  
  
void foo(int * a)  
{return 0;}
```

(1) En la llamada, se envía la **dirección** del argumento usando el operador **&**

(2) En la función invocada, se usa el operador **\*** para que una **variable puntero local** apunte a la **dirección** recibida

# (sin) Punteros: Ejemplo → Parámetro por valor (1)

---

```
int main()
```

```
{
```

```
    int number = 5;
```

```
    number = cubeByValue( number );
```

```
}
```

number

5

```
int cubeByValue( int n )
```

```
{
```

```
    return n * n * n;
```

```
}
```

n

undefined

**Paso 1:** Antes de que **main** llame a **cubeByValue**

# (sin) Punteros:

## Ejemplo → Parámetro por valor

### (2)

---

```
int main()
{
    int number = 5;

    number = cubeByValue( number );
}
```

number

5

```
int cubeByValue( int n )
{
    return n * n * n;
}
```

n

5

**Paso 2:** Después de que **cubeByValue** recibe la llamada

# (sin) Punteros:

## Ejemplo → Parámetro por valor


### (3)

---

```
int main()
{
    int number = 5;

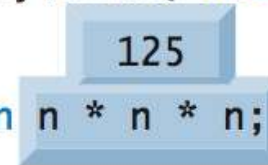
    number = cubeByValue( number );
}
```

number

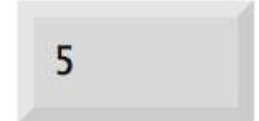


```
int cubeByValue( int n )
{
    return n * n * n;
}
```

125



n



**Paso 3:** Después de que **cubeByValue** eleva el parámetro **n** al cubo y antes de que retorne al **main**

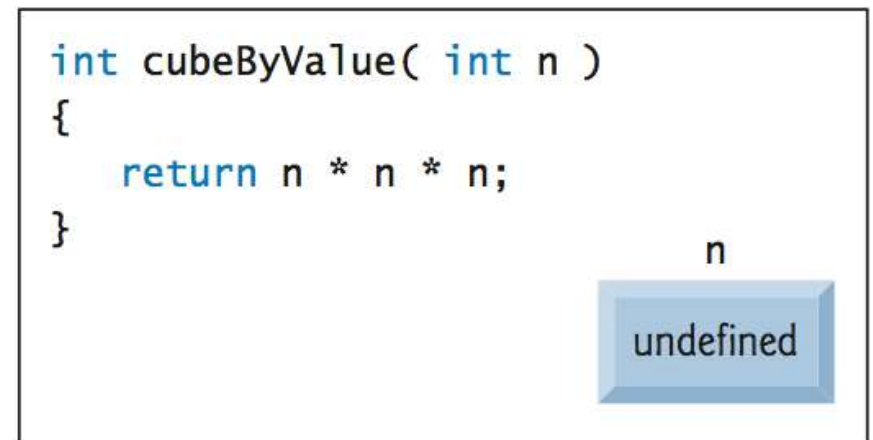
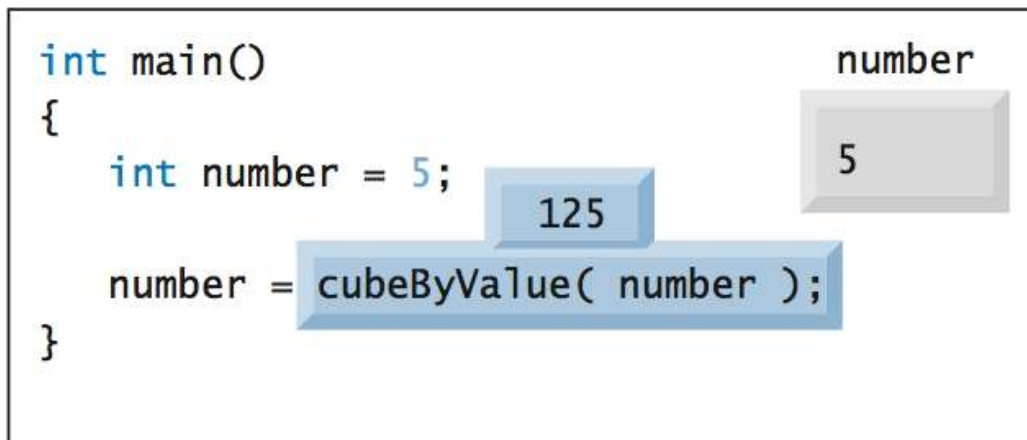


# (sin) Punteros:

## Ejemplo → Parámetro por valor

### (4)

---



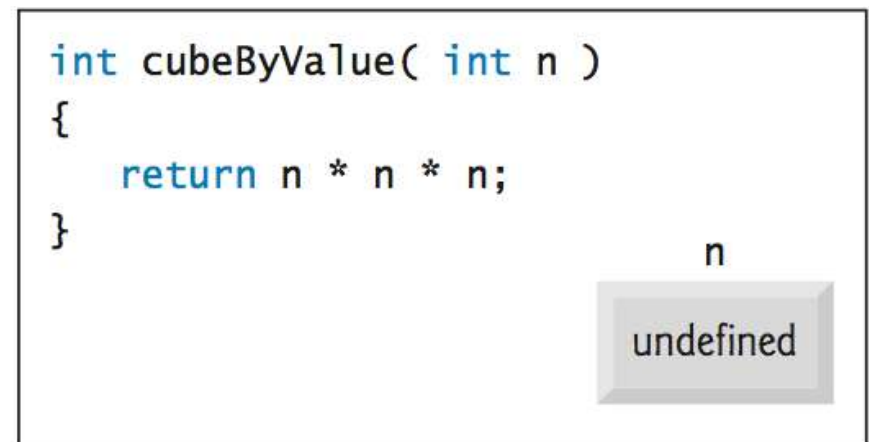
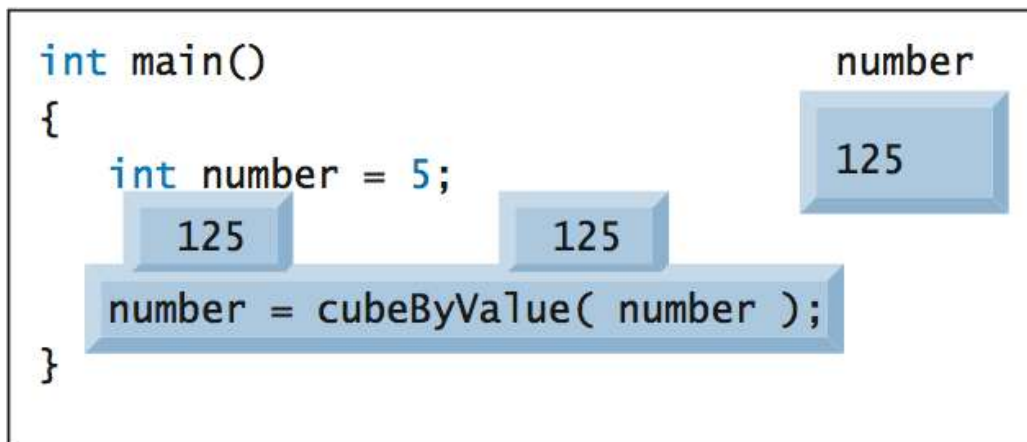
**Paso 4:** Después de que **cubeByValue** retorna a **main** y antes de asignar el resultado a **number**

# (sin) Punteros:

## Ejemplo → Parámetro por valor

### (5)

---



**Paso 5:** Después de que **main** completa la asignación a **number**

# (con) Punteros:

## Ej. → Parámetro por Referencia

### (1)

---

```
int main()
{
    int number = 5;

    cubeByReference( &number );
}
```

number

5

```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

undefined

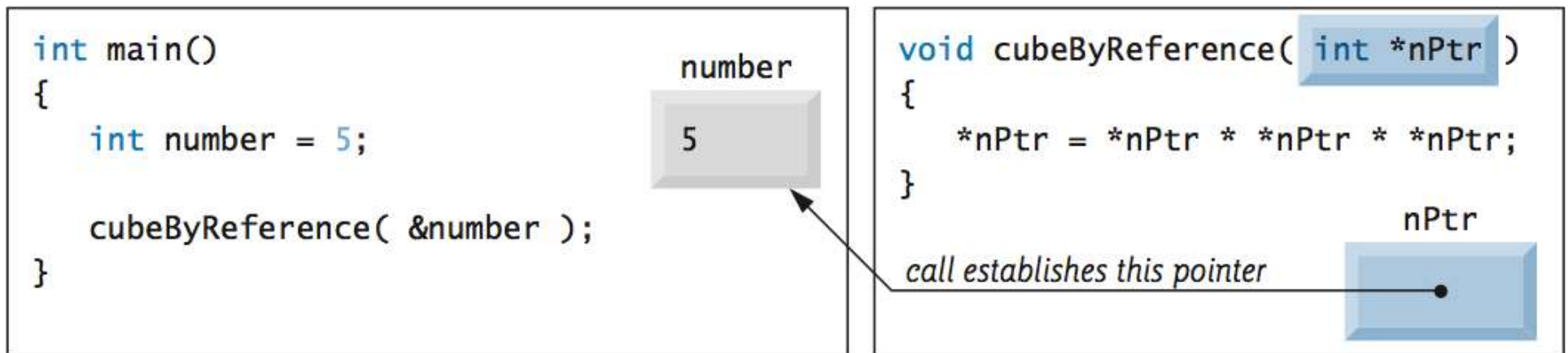
**Paso 1:** Antes de que **main** llame a **cubeByReference**

# (con) Punteros:

## Ej. → Parámetro por Referencia

### (2)

---



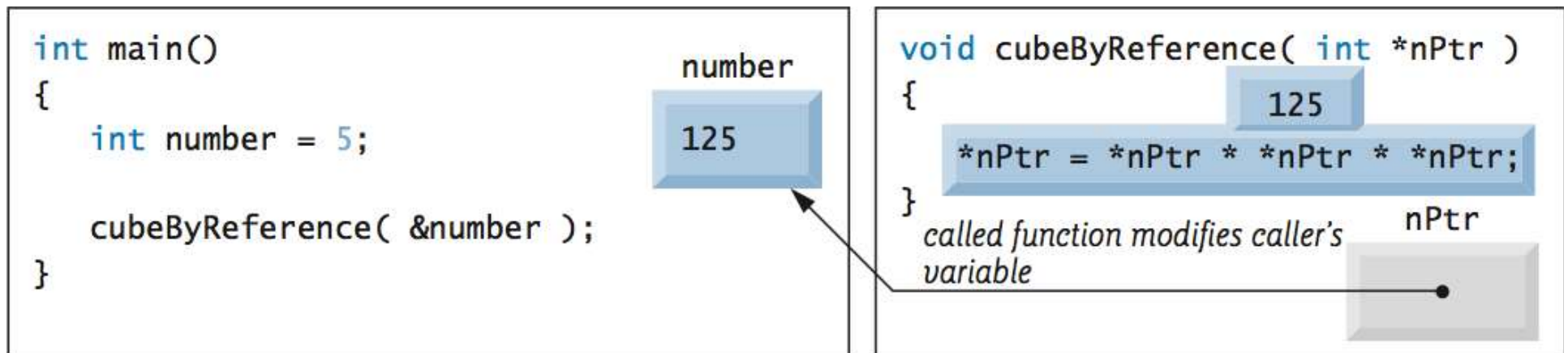
**Paso 2:** Después de que **cubeByReference** recibe la llamada y antes de que eleve **\*nPtr** al cubo

# (con) Punteros:

## Ej. → Parámetro por Referencia

### (3)

---



**Paso 2:** Después de que `*nPtr` es elevado al cubo y antes de que el control del programa retorne a `main`

# Punteros: Aritmética

---

Los punteros pueden ser **incrementados** o **decrementados**

```
countPtr++;  
countPtr--;
```

A los punteros se les puede **sumar** o **restar** un **valor entero** `ptr +=`

```
7;    ptr = ptr + 7;  
ptr -= 7;    ptr = ptr - 7;
```

Los punteros pueden ser **sumados** o **restados** entre sí, siempre y cuando apunten al **mismo tipo de dato**

```
int a = 5, b = 7, c;  
int *ptrA = &a, *ptrB = &b;  
c = ptrA - ptrB;
```

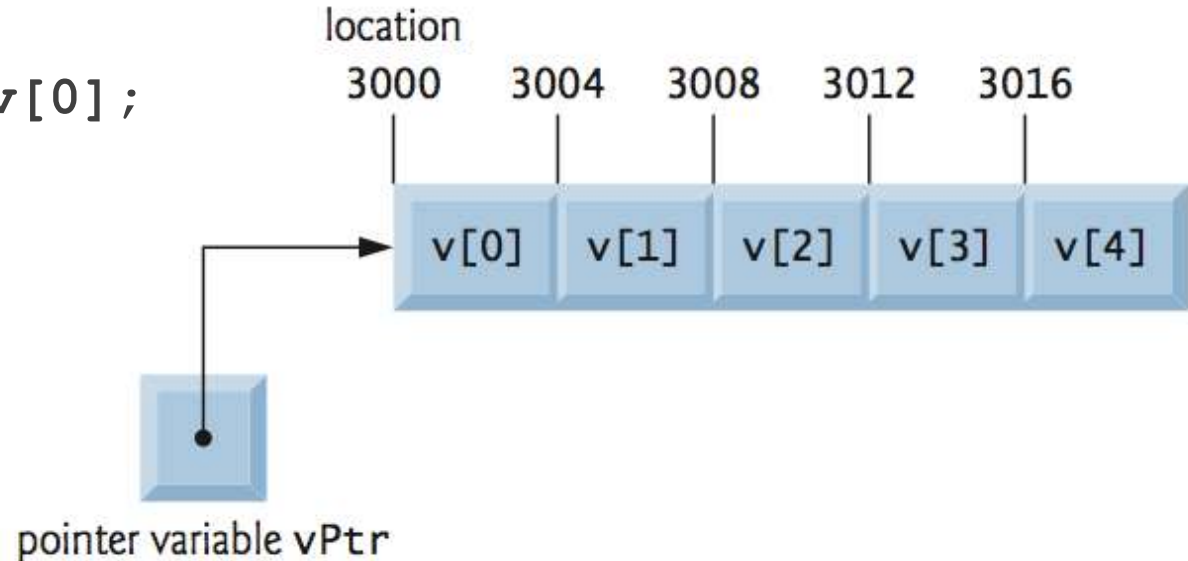
# Punteros: Aritmética y su aplicación en Arreglos

---

```
int v[5] = {0,1,2,3,4};
```

```
int *vPtr = v;
```

```
//int *vPtr = &v[0];
```



# Punteros: Aritmética y su aplicación en Arreglos

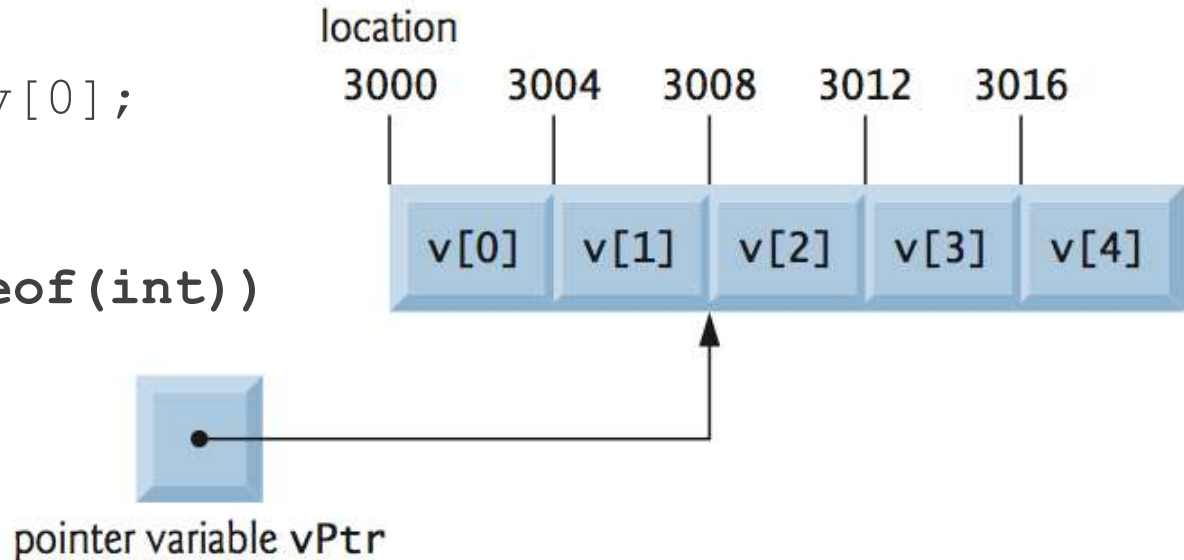
```
int v[5] = {0,1,2,3,4};
```

```
int *vPtr = v;
```

```
//int *vPtr = &v[0];
```

```
vPtr += 2;
```

```
 //(3000 + 2*sizeof(int))
```





# Punteros: Memoria Dinámica

---

Permite reservar memoria del **tamaño que se desee** en el **momento en que se desee**.

Se **reserva espacio** de la **memoria dinámica**.

## Memoria dinámica en C:

**malloc** → Reserva de memoria

**free** → Liberación de memoria

**realloc** → `malloc + memcpy + free`

# Punteros: Memoria Dinámica (malloc)

---

Veamos el siguiente código:

```
#include <malloc.h>           // También se puede emplear stdlib.h o alloc.h

int main ()
{ int * pt;

    pt = (int *) malloc( sizeof(int) ); // se asigna a pt la dirección de un
                                         // bloque de memoria del tamaño de un int
    *pt = 37;                        // ya se puede se asignar valores a la variable referenciada
}
```

# Punteros:

## Memoria Dinámica (Arreglos dinámicos)

---

```
#include <stdio.h>
#include <malloc.h>
```

```
int main ()
{ int * pt; int numElem;
```

```
printf("Ingrese el número de datos del arreglo: ");
scanf("%d", &numElem);
```

```
pt = (int *) malloc( numElem * sizeof(int) ); // se asigna a pt la dirección de un
//bloque de memoria del tamaño de un arreglo de tipo int de tamaño numElem
```

```
pt[2] = 177; // se puede emplear pt como un arreglo común
*pt = 88; // es equivalente a hacer pt[0] = 88;
*(pt+2) = 92; // es equivalente a hacer pt[2] = 92;
}
```

# Punteros:

## Memoria Dinámica (free)

---

**void free** (dirección del bloque);

Por ejemplo:

```
#include <stdio.h>
#include <malloc.h>
int main (void)
{ int * pt;
  int numElem;

  printf("Ingrese el número de datos del arreglo: ");
  scanf("%d", &numElem);
  pt = (int *) malloc( numElem * sizeof(int) );
  *pt = 88;
  free(pt); // el bloque fue liberado
```

# Punteros:

## Memoria Dinámica (realloc)

---

Esta función permite trasladar la información contenida en un bloque de memoria a otro bloque de mayor tamaño. La función devuelve la dirección del byte de inicio del bloque y libera la memoria dada al bloque original. El prototipo de la esta función se muestra a continuación:

```
void * realloc (dirección del bloque original, nuevo tamaño);
```

Si por alguna razón no se pueda encontrar un bloque del tamaño pedido, la función **realloc** devuelve **NULL** y el bloque de memoria original no es liberado.

# Referencias

---

DEITEL, P; DEITEL, H. **C++ How To Program**. 9na edición. USA: Prentice Hall, 2015. ISBN 978-0-13-337871-9.

STROUSTRUP, B. **The C++ Programming Language**. 4ta edición. USA: Addison-Wesley, 2013. ISBN 978-0-321-56384-2.

**Presentación original desarrollada por el Prof. Arturo Oncevay y Miguel Guanira**