Programación Orientada a Objetos

Unidad 1: Punteros

FICH UNL
Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas

2019
Pablo Novara

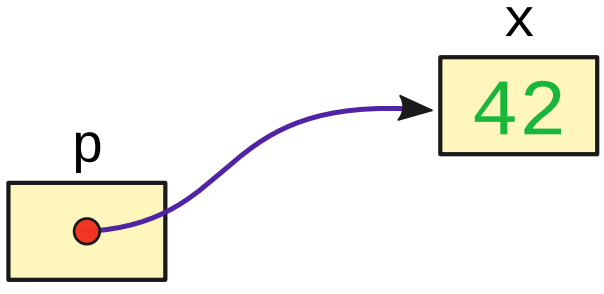
¿QUÉ #@*$&! ES UN PUNTERO?

```
int x = 42;
int *p;
p = &x;
```

variable tipo int: pedacito de memoria para guardar un entero

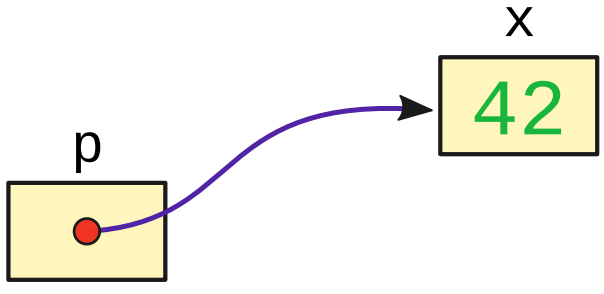
puntero a int: variable que guarda la dirección de memoria de algún entero

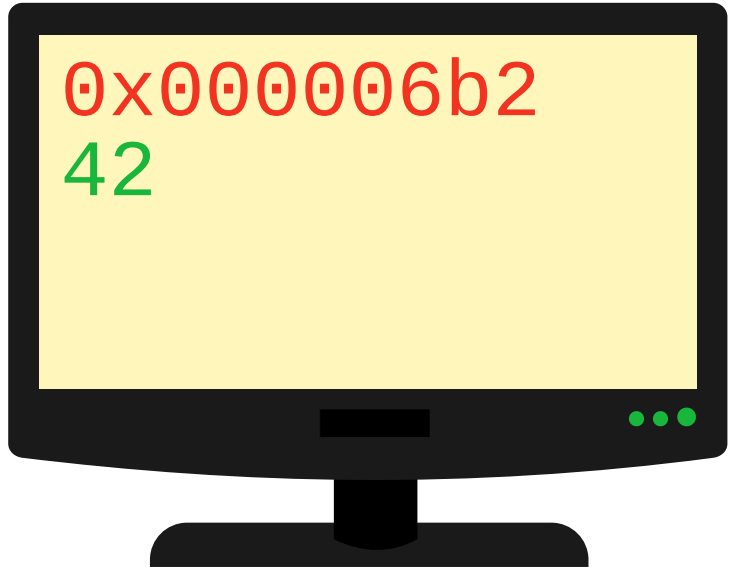
p toma la dirección de memoria de x



¿QUÉ #@*$&! ES UN PUNTERO?

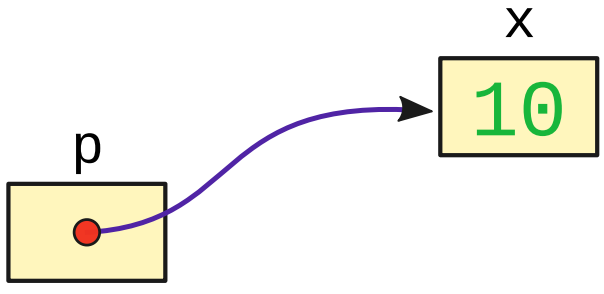
```
int x=42;
int *p;        // int* significa "puntero a int"
p = &x;        // &x significa "la dirección de la variable x"
cout << p << endl;
cout << *p; // *p significa "el dato que está en la dirección p"
```

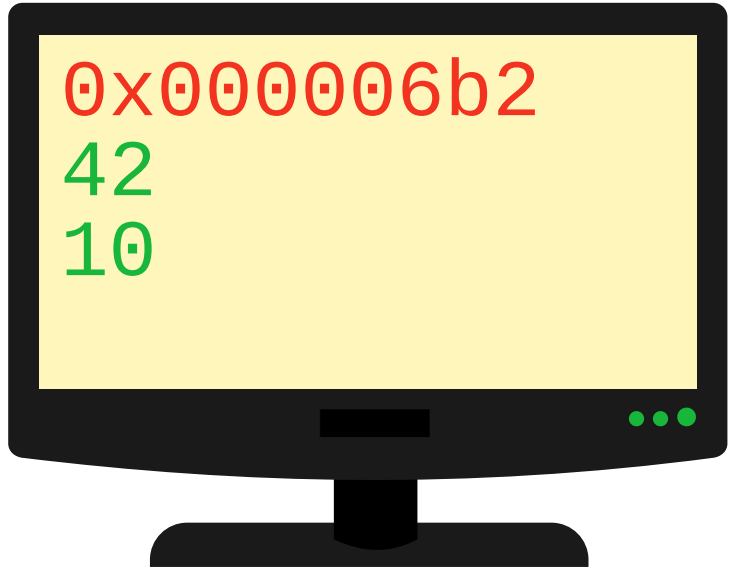




¿QUÉ #@*$&! ES UN PUNTERO?

```
int x = 42;
int *p;         // int* significa "puntero a int"
p = &x;         // &x significa "la dirección de la variable x"
cout << p << endl;
cout << *p; // *p significa "el dato que está en la dirección p"

*p = 10;
cout << endl << x;
```





OPERADORES: & Y *

- En un tipo:

```
int& x = a; // x es alias de a
```

```
int* p; // p es un nuevo puntero
```

solo se define un puntero pero todavía no apunta a ningún lado

- Sobre una variable o expresión:

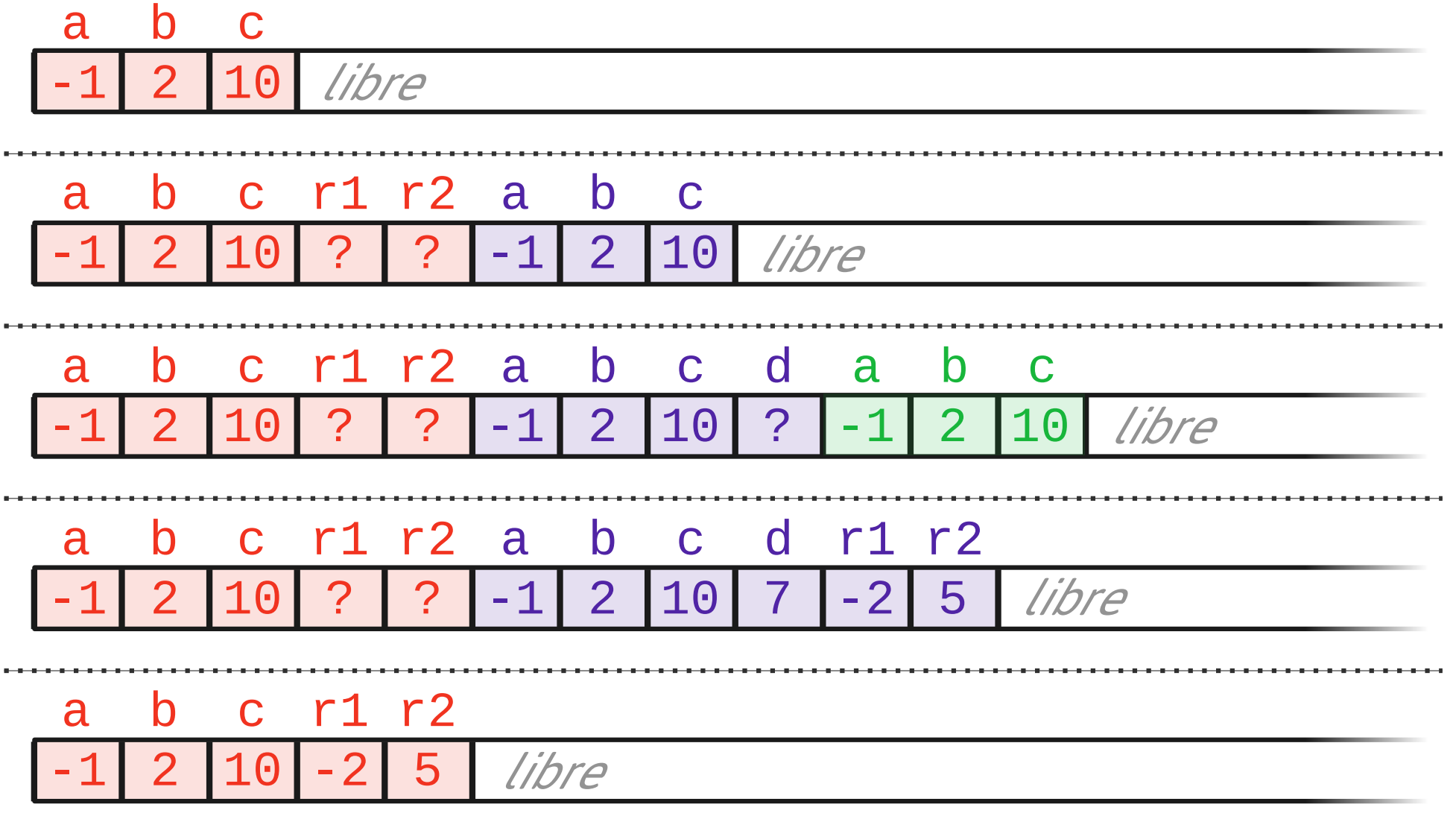
```
cout << &a; // la dirección de a
```

```
cout << *p; // el dato apuntado por p
```

EJEMPLO 1: EL STACK

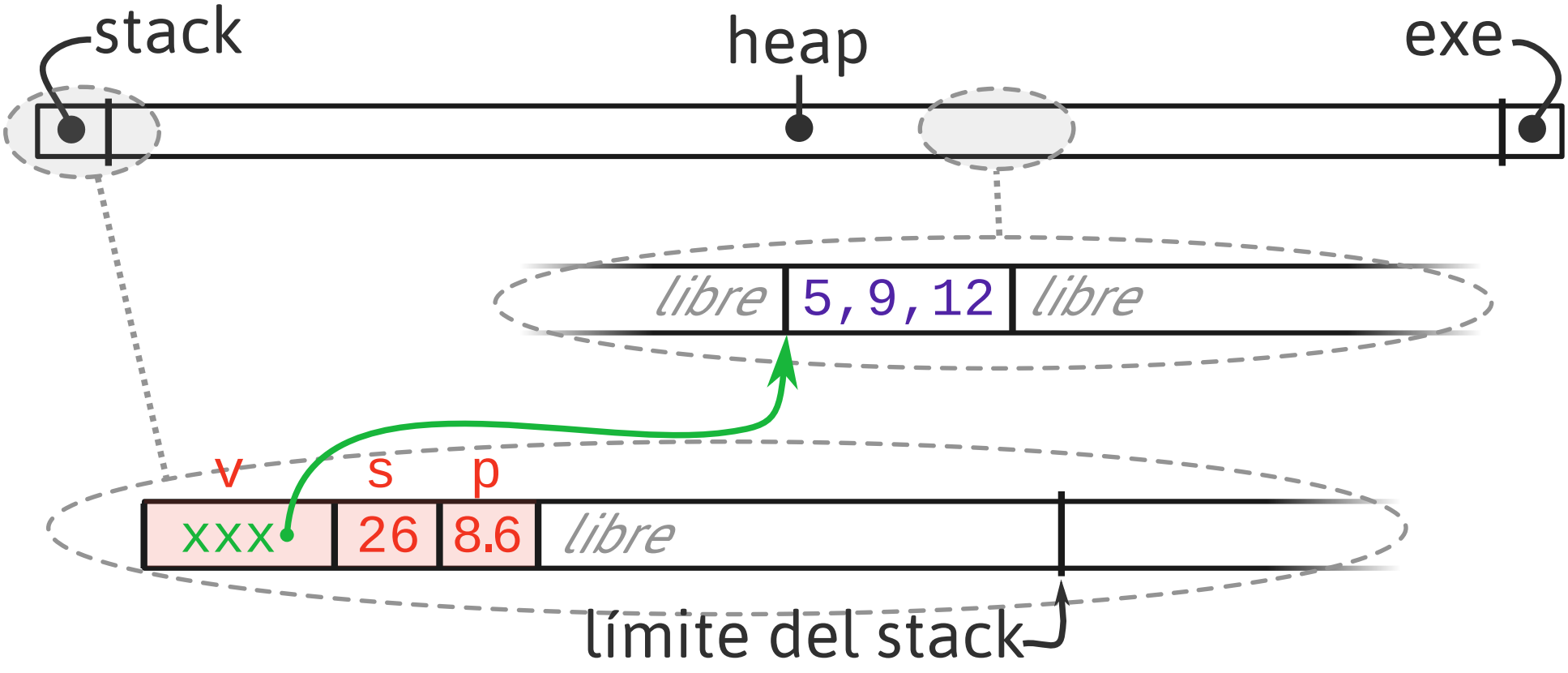
```
int main() {
    float a, b, c;
    cin >> a >> b >> c;
    float r1, r2;
    tie(r1,r2) = raices(a,b,c);
    cout << r1 << endl << r2 << endl;
}

tuple<...> raices(float a, float b, float c) {
    float d = discrim(a,b,c);
    float r1 = (-b+sqrt(d))/(2*a);
    float r2 = (-b-sqrt(d))/(2*a);
    return make_tuple(r1,r2);
}

float discrim(float a, float b, float c) {
    float d = b*b-4*a*c;
    return d;
}
```

EJEMPLO 1: EL STACK



EJEMPLO 2: EL HEAP

```
int main() {
    vector<float> v = { 5, 9, 12 };
    float sum = suma_vector(v);
    float prom = sum / v.size();
    v.push_back(prom);
    ...
```



GESTIÓN DINÁMICA DE LA MEMORIA

Permite:

- Aprovechar toda la memoria disponible para guardar grandes volúmenes de datos.
- Almacenar datos cuyo tamaño se desconoce en tiempo de compilación.
- Alterar el ciclo de vida de una variable (que viva más o menos que su scope).

OPERADORES new Y delete

Para solicitar memoria dinámica se utiliza `new`:

```
puntero = new tipo;
```

reserva un bloque de memoria del tamaño adecuado y retorna un puntero al mismo.

Para liberar esa memoria se utiliza `delete`:

```
delete puntero;
```

```
int *ptr = new int;  // obtiene memoria para un entero
*ptr = 42;           // guarda 42 en esa memoria
cout << *ptr;        // muestra el contenido: 42
delete ptr;          // libera la memoria
```

*esta memoria **no** se libera automáticamente, es un error no hacer nunca el `delete`*

ARREGLOS ESTÁTICOS EN C/C++

Los arreglos estáticos tienen tamaños...

- **constantes**: no pueden cambian su tamaño durante la ejecución
- **definidos en tiempo de compilación**: no pueden depender de datos que se ingresan durante la ejecución

Se declaran especificando sus dimensiones entre corchetes luego del nombre:

```
int v[10]; // arreglo lineal de 10 enteros
float m[3][5]; // matriz de 3x5 flotantes
```

una matriz de 3x5 floats es en realidad un arreglo lineal de 3 elementos, donde cada elemento es a su vez un arreglo lineal de 5 floats

RESUMEN DE TIPOS

- variable "común" de tipo float:
  - **`float`** `Ej: float var;`
- referencia/alias a una variable de tipo float:
  - **`float&`** `Ej: float &ref;`
- puntero a una variable de tipo float:
  - **`float*`** `Ej: float *ptr;`
- arreglo de N(cte) elementos de tipo float:
  - **`float[N]`** `Ej: float vec[10];`

ARREGLOS DINÁMICOS EN C/C++

Se crean con el operador **new[]**
y se destruyen con el operador **delete[]**

```
int n;
cin >> n;
float *v = new float[n];
for(int i=0; i<n; i++) cin >> v[i];
for(int i=0; i<n; i++) cout << v[i] << endl;
delete [] v;
```

- El tamaño puede provenir de una variable...
  - ...pero luego no se lo puedo "preguntar" a v
- El puntero apunta al 1er elemento

EJEMPLOS

1. Escriba una función que genere, inicialice y retorne un arreglo de N números aleatorios entre 1000 y 1500.

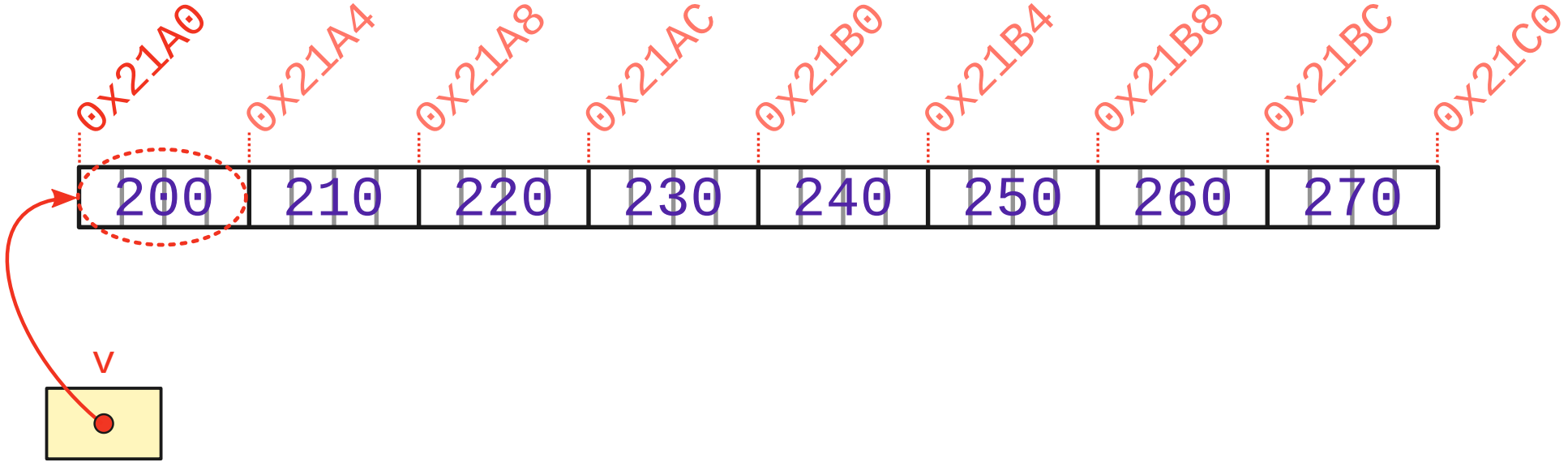
2. Escriba un programa cliente para dicha función que permita ingresar N y muestre el arreglo generado.

ARITMÉTICA DE PUNTEROS

Dado el código...

```
int v[8] = { 200, 210, 220, 230,
             240, 250, 260, 270 };
```

...y suponiendo que los datos del arreglo se almacena a partir de la posición `0x21A0`...



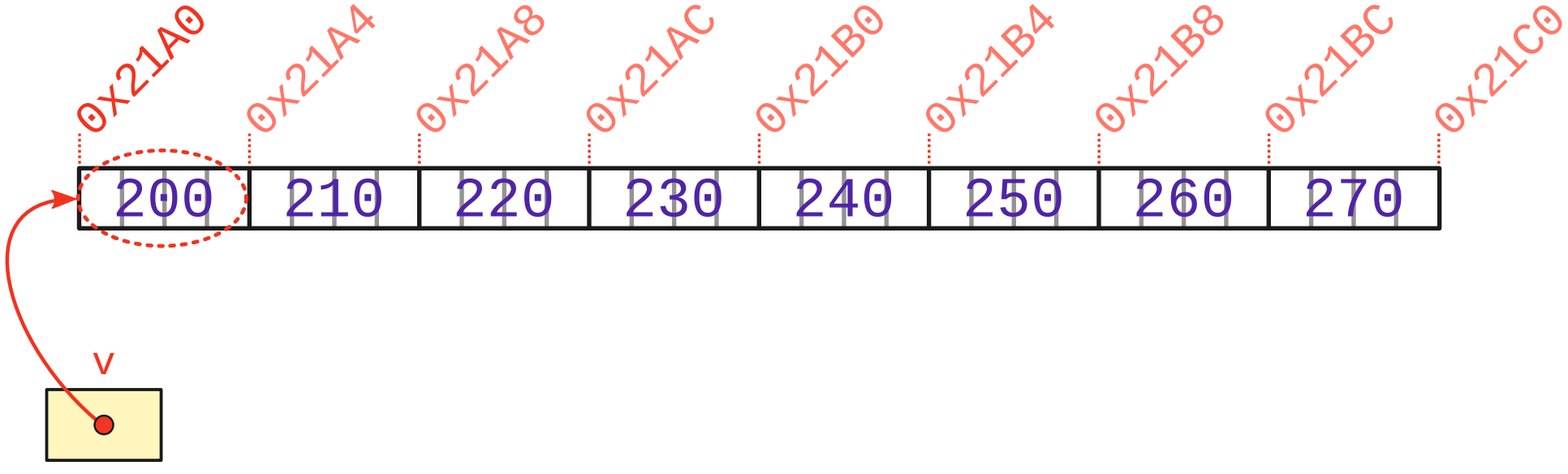
ARITMÉTICA DE PUNTEROS

¿Cuánto vale v?

0x21A0

¿y *v?

*0x21A0 = 200



ARITMÉTICA DE PUNTEROS

¿Cuánto vale `*v + 3`?

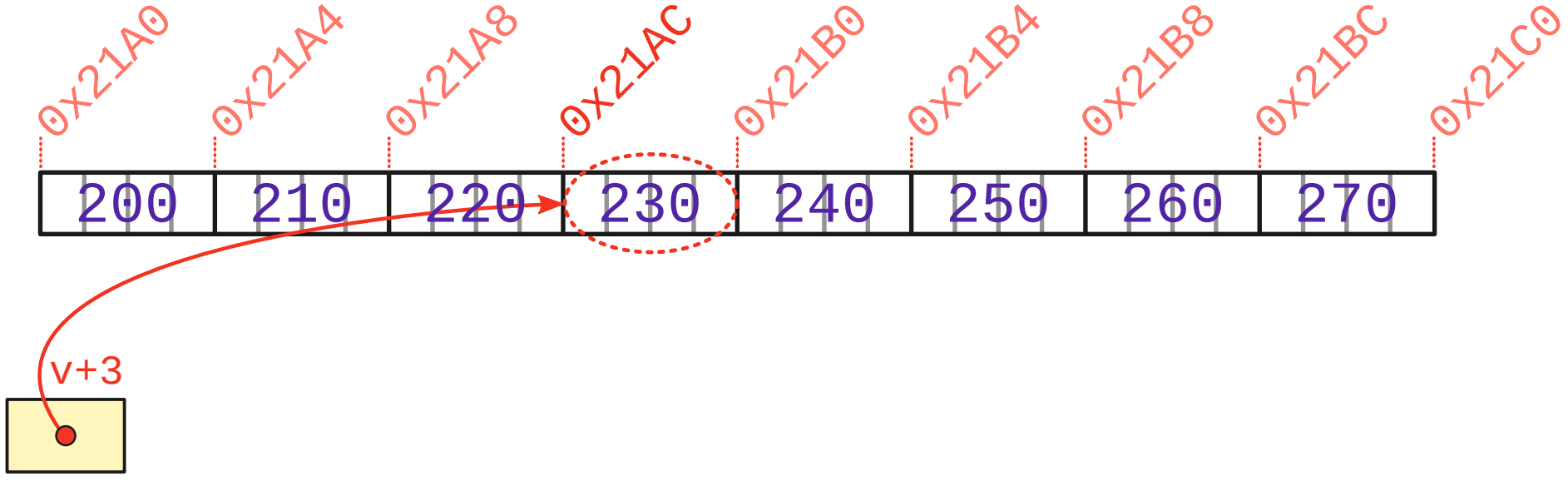
`(*0x21A0)+3 = 200+3 = 203`

¿y `*(v+3)`?

`*(0x21A0 + 3x4) = *(0x21AC) = 230`

dirección del inicio del vector (puntero)

tamaño de un elemento (sizeof(int) = 4 bytes)

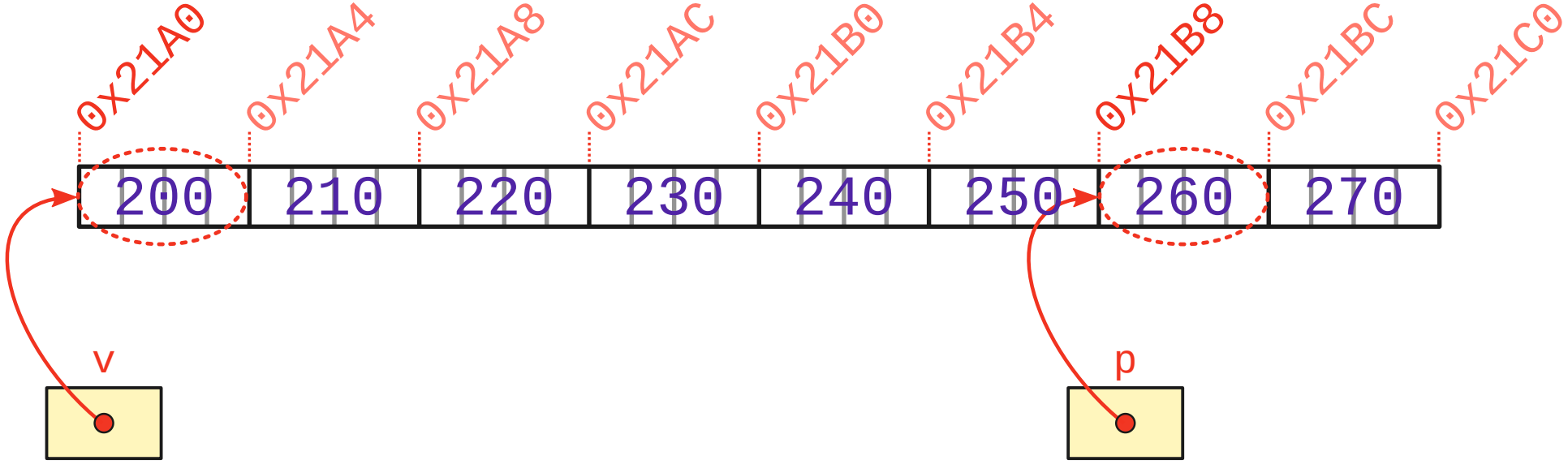


ARITMÉTICA DE PUNTEROS

¿A qué posición del arreglo apunta p?

(0x21B8-0x21A0)/4 = 24/4

p-v = 6



EL PUNTERO NULO

`nullptr` corresponde una dirección de memoria no válida (la dirección 0).

- Es recomendable:
  - inicializar con `nullptr` todo puntero que no se inicialice con una dirección válida.
  - asignar `nullptr` a un puntero cuando la dirección que guarda deja de ser válida.

En C++98/03 se utilizaba `NULL` en lugar de `nullptr`. Desde C++11 en adelante ambos son válidos, y se recomienda preferir `nullptr`.

EJEMPLOS

3. Escriba una función que permita buscar un elemento en un arreglo y retorne un puntero al mismo si lo encuentra, o el puntero nulo en caso contrario.

4. Escriba un programa cliente para probar dicha función y mostrar en qué posición del arreglo se encuentra el elemento buscado.

HEAP VS. STACK

Variables en el stack:

```
{
    ...
    int x; // se crea un entero
    ...  usar x ...
} // se destruye automáticamente
```

El ciclo de vida está dado por el scope.

Variables en el heap:

```
int *p = new int; // se crea el nuevo int
...  usar *p ...
delete p; // se destruye/libera la memoria
```

El ciclo de vida es arbitrario, pero genera la responsabilidad del delete

HEAP VS. STACK

Arreglos en el stack:

```
{
    ...
    int v[10]; // se crea un arreglo
    ...
} // se destruye automáticamente
```

El tamaño es fijo (cte. en tiempo de compilación).

Arreglos en el heap:

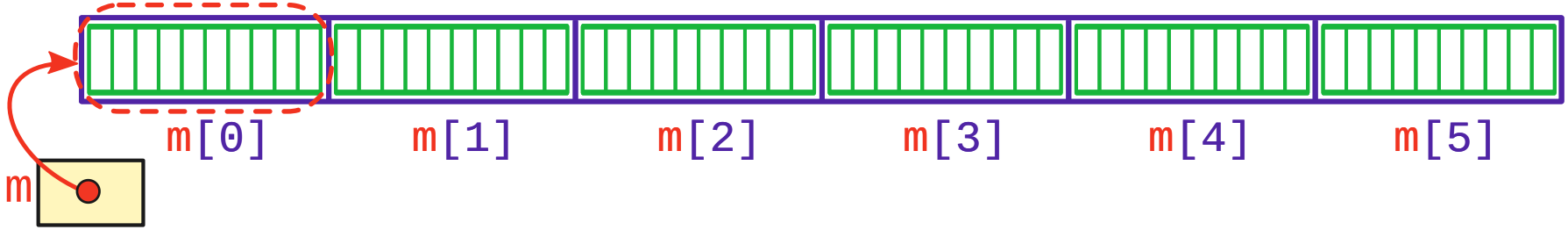
```
int *v = new int[n]; // se crea el nuevo int
...
delete [] v; // se destruye/libera la memoria
```

El tamaño se define en tiempo de ejecución.

PUNTEROS Y MATRICES ESTÁTICAS

```
int m[6][10];
```

Ambas dimensiones fijas, 60 ints **contiguos**.



m es de tipo "puntero a arreglo de 10",

```
int (*p)[10] = m;
```

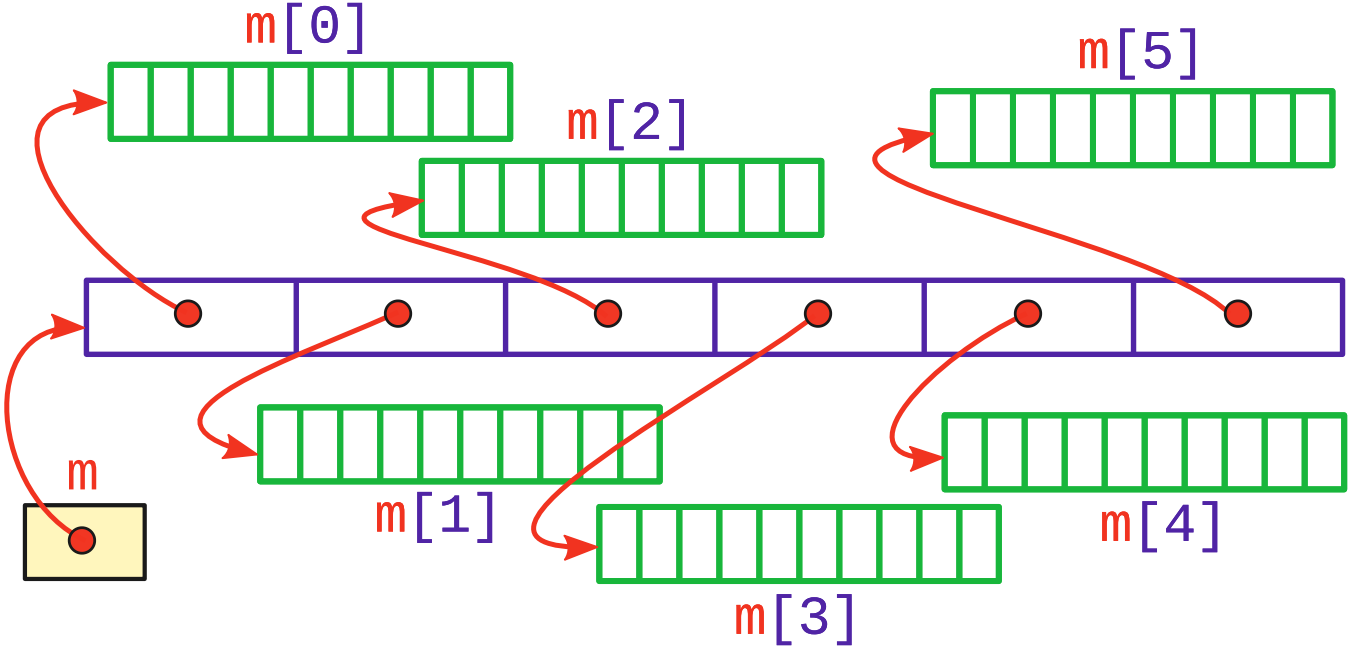
Para acceder a un elemento:

```
m[i][j] = *( *(m+i)+j )
```

PUNTEROS Y MATRICES DINÁMICAS

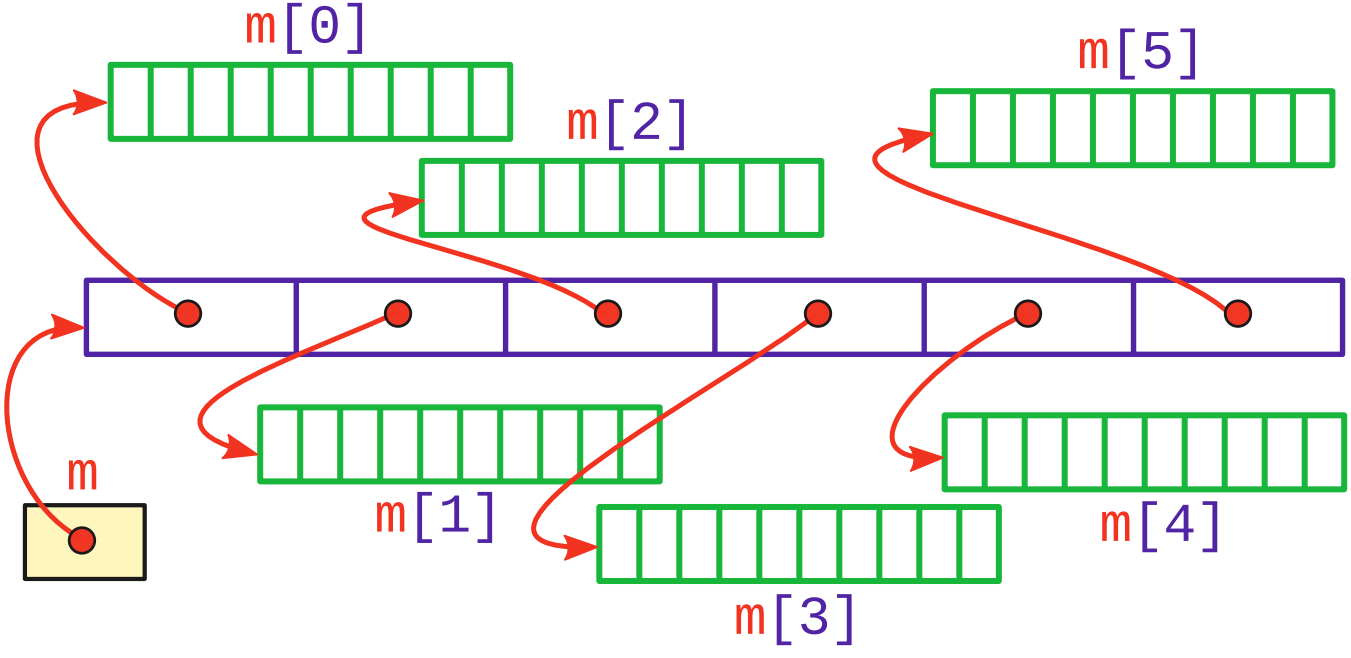
- Ambas dimensiones dinámicas
- Elementos **no contiguos**
- `m` es de tipo "puntero a puntero a entero"

```
int **m = ???;
```



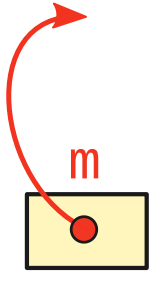
PUNTEROS Y MATRICES DINÁMICAS

```
int **m = new int*[6];
for(int i=0; i<6; i++)
    m[i] = new int[10];
```



PUNTEROS Y MATRICES DINÁMICAS

```
int **m = new int*[6];
for(int i=0; i<6; i++)
    m[i] = new int[10];
...
for(int i=0; i<6; i++)
    delete [] m[i];
delete [] m;
```



RESUMEN DE TIPOS 2

- referencia/alias a una variable de tipo float:
  - **`float&`** `Ej: float &ref;`
- puntero a una variable de tipo float:
  - **`float*`** `Ej: float *ptr;`
- arreglo de 8 elementos de tipo float:
  - **`float[8]`** `Ej: float vec[10];`
- arreglo de 8 elementos de tipo puntero a float:
  - **`float*[8]`** `Ej: float *vp[10];`
- puntero a un arreglo de 8 elementos de tipo float:
  - **`float(*)[8]`** `Ej: float (*pv)[10];`

EL OPERADOR ->

```
struct Pelicula {
    string titulo, genero;
    int anio, duracion;
};

Pelicula *una_peli = new Pelicula;
```

¿Cómo acceder al título de la película apuntada?

- **una_peli** es un ptr. a una var. de tipo Pelicula
- ***una_peli** es una variable de tipo Pelicula
- **(*una_peli).titulo** es el título de la Pelicula
- **una_peli->titulo** también es el título