

# Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

¿Cómo sería un programa recursivo en C++ para calcular el N-ésimo número de Fibonacci?

Donde  $F_0 = 0$  ,  $F_1 = 1$  ,  $F_n = F_{n-1} + F_{n-2}$  para  $n \geq 2$

Hagámoslo juntos.....



Donde  $F_0 = 0$ ,  $F_1 = 1$ ,  $F_n = F_{n-1} + F_{n-2}$  para  $n \geq 2$

Un algoritmo recursivo sería:

```
/* *
 *   fib(n):
 *       n debe ser un entero no negativo
 *       Devuelve el número de Fibonacci número n
 * */
long  fib( int n )
{   if( n <= 1 )   // casos base n=0 y n=1
        return n;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```

este algoritmo es exponencial función de n

Un algoritmo iterativo sería:

```
long fibonacci_itr( int n )
```

```
{ // n debe ser >= 0
```

```
    int fn_1 = 1, fn_2 = 0, fib = 0 ;
```

```
    if (n==0 || n==1) return n;
```

```
    for (int i = 2; i<=n; i++) {
```

```
        fib = fn_1 + fn_2;  fn_2 = fn_1;  fn_1 = fib;
```

```
    }
```

```
    return fib;
```

```
}    Mejor en tiempo (n-1 sumas) y espacio que el recursivo....
```

**Se debe evitar la recursión:**

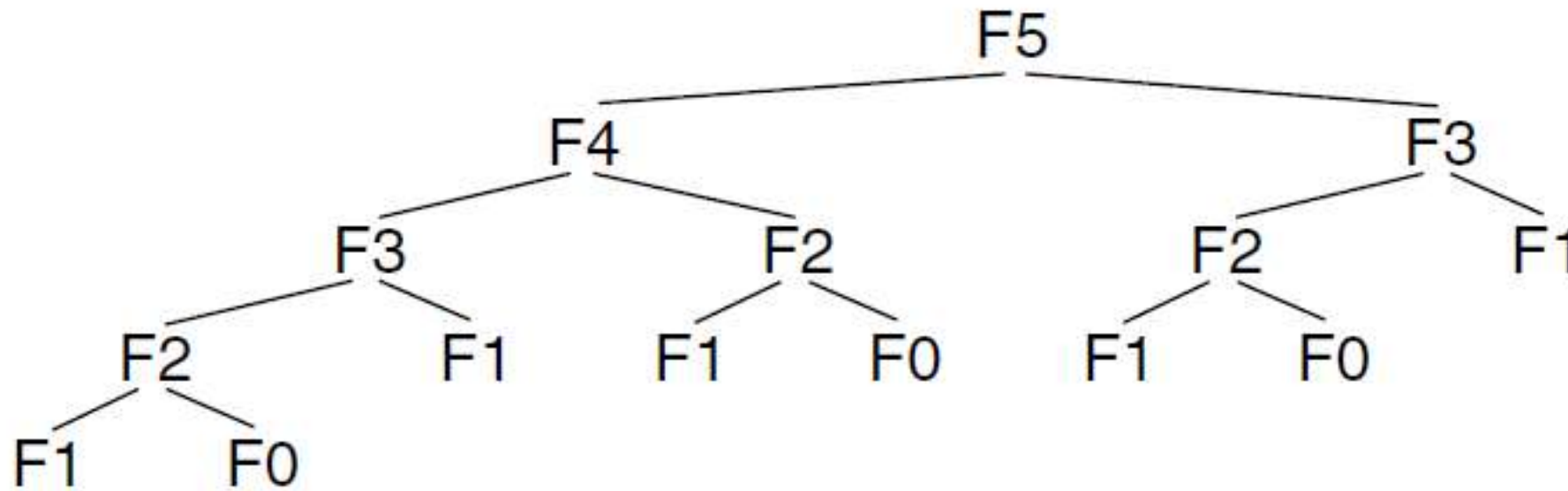
**Cuando los sub-problemas que se resuelven se repiten, es decir, tienen los mismos datos!! Esto puede aumentar drásticamente el tiempo de ejecución.**

Ejemplo: Cálculo de los **números de Fibonacci**  $F_0, F_1, F_2, \dots, F_n, \dots$

Donde  $F_0 = 0$  ,  $F_1 = 1$  ,  $F_n = F_{n-1} + F_{n-2}$  para  $n \geq 2$

$F_0 = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, F_{11} = 89, \dots$

Veamos el árbol de llamadas que realiza el algoritmo recursivo para  $N = 5$ :



Note que **F3** se calcula 2 veces, **F2** y **F0** se calculan 3 veces, **F1** se calcula 5 veces

En mi computador, calcular F50 tardó 98 segundos!! Algo absurdo porque si calculamos iterativamente son sólo 49 sumas!!!

El iterativo tarda menos de un milisegundo

Se puede demostrar que el algoritmo recursivo tiene un tiempo de ejecución exponencial.

# ANÁLISIS DEL ALGORITMO FIB:

## **$T(N)$ es el tiempo de ejecución para $N$**

```
public static long fib( int n )  
{   if( n <= 1 )  
    return 1;  
    else  
    return fib( n - 1 )  
        + fib( n - 2 );  
}
```

-  $T(0) = T(1) = 1$ , un tiempo constante.

- Si  $N > 2$ ,  $T(N) = T(N-1) + T(N-2) + 3$ , el 3 por la condición del IF, la suma de dos números (los que devuelven las llamadas recursivas) y el return del resultado

- Como  $\text{fib}(N) = \text{fib}(N-1) + \text{fib}(N-2)$  se puede mostrar que  $T(N) \geq \text{fib}(N)$  y se sabe que  $\text{fib}(N) \geq (3/2)^N$

-Tenemos que  $T(N) = \Omega ((3/2)^N)$  exponencial !!

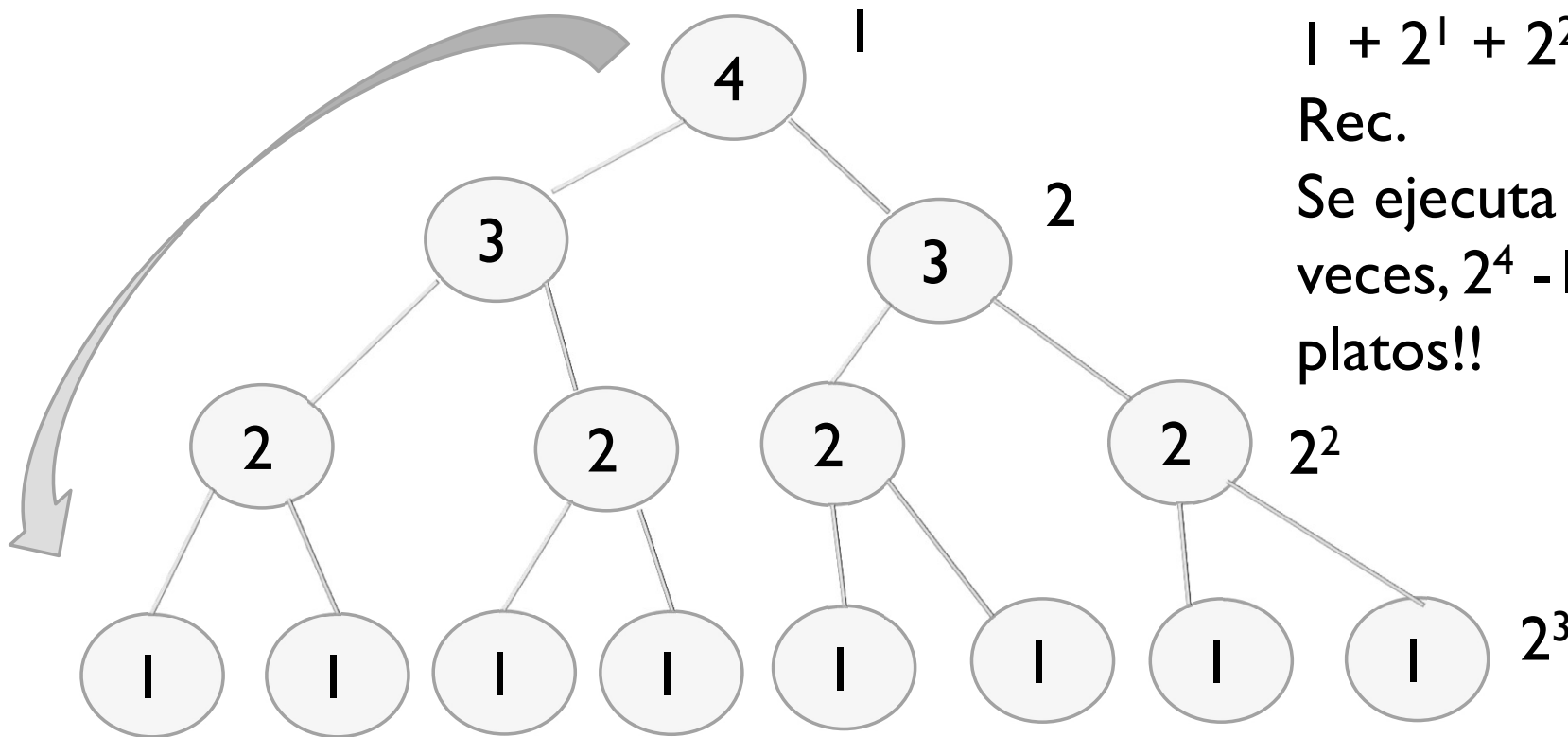


## Veamos el árbol de llamadas que genera el algoritmo recursivo de las torres de Hanoi para $N = 4$ platos:

En cada nodo se mueve un plato

$$1 + 2^1 + 2^2 + 2^3 = 2^4 - 1 = 16 - 1 = 15 \text{ act. Rec.}$$

Se ejecuta un número exponencial de veces,  $2^4 - 1$  función del número de platos!!



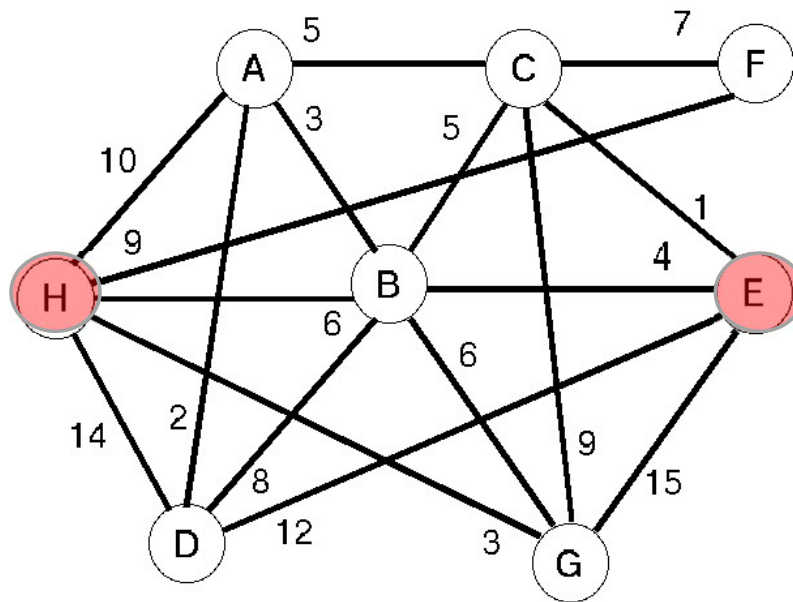
**RECALQUEMOS.....**Una solución recursiva se sustenta en dos hechos fundamentales:

1. Existe un conjunto de **casos bases (problemas base del mismo tipo)** que no se resuelven de manera recursiva sino directa.
2. **Los sub-problemas “más pequeños”** tienen menos datos. Lo importante es que **los sub-problemas van convergiendo en forma estricta a algunos de los casos base**. Cuando decimos en forma estricta es para evitar que la solución del problema quede expresada en función de la solución del mismo problema con los mismos datos y se crea así una **solución cíclica**, que se devuelve al problema original y no lo resuelve.

En el caso del factorial vemos que el subproblema  $(N-1)!$  Es más pequeño pues converge al caso base, en este caso  $0!=1$

## Un ejemplo de “solución” recursiva cíclica y que por lo tanto no resuelve el problema:

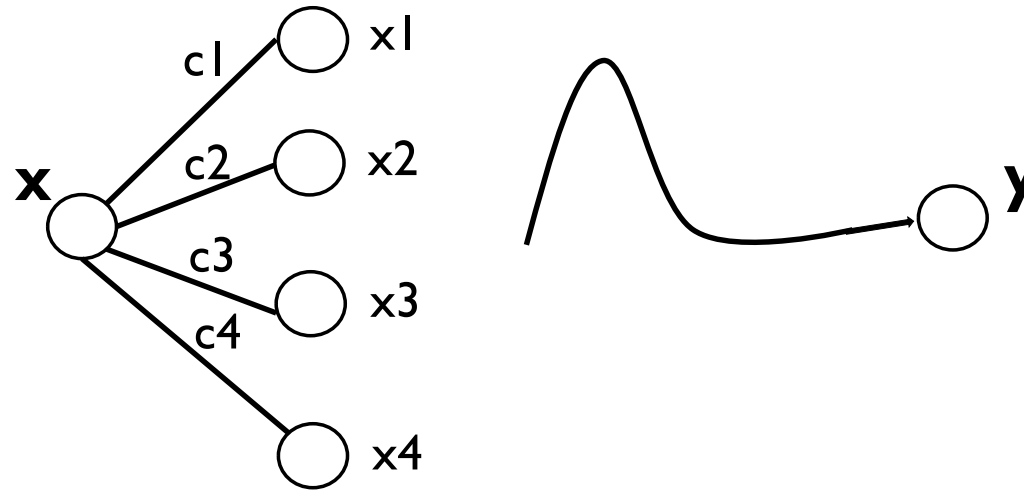
- Hallar el camino de menor costo desde un punto a otro en una red



Hallar un camino de menor costo desde H a E

El costo de un camino es la suma de los costos de los arcos que componen al camino

- Sea **x** el punto de partida e **y** el punto de llegada. Podemos formular la siguiente solución recursiva:



Si **Cmin** significa costo de un camino mínimo. Una solución recursiva errada:

$$\text{Cmin de } x \text{ a } y = \min \{ c1 + \text{Cmin}(x1, y), c2 + \text{Cmin}(x2, y), c3 + \text{Cmin}(x3, y), c4 + \text{Cmin}(x4, y) \}$$

Note que  $\text{Cmin}(x1, y)$  se expresa en función de  $\text{Cmin}(x, y)$  y obtenemos un ciclo:

$$\text{Cmin de } x1 \text{ a } y = \min \{ c1 + \text{Cmin}(x, y), \dots \}$$

La técnica de resolución de problemas **DIVIDE-AND-CONQUER** (dividir y conquistar) se basa en recursividad.

Para resolver un problema por **DIVIDE AND CONQUER** hacemos:

- 1) Dividir:** resolver subproblemas más pequeños del mismo tipo
- 2) Conquer:** componer la solución del problema en base a la solución de los subproblemas resueltos

Un algoritmo en el cual **al menos dos subproblemas** son resueltos recursivamente son llamados **algoritmos divide-and-conquer**. Por lo que el algoritmo del **factorial** no es considerado divide-and-conquer, sino recursión de cola. Mientras que el de las **Torres de Hanoi** es un algoritmo divide-and-conquer

## Resolvamos el siguiente problema usando la técnica divide-and-conquer:

Encontrar en una secuencia de números una sub-secuencia contigua con la suma máxima.

Ejemplo:

-2	4	-3	5
----	---	----	---

las sub-secuencias contiguas de la secuencia anterior son:

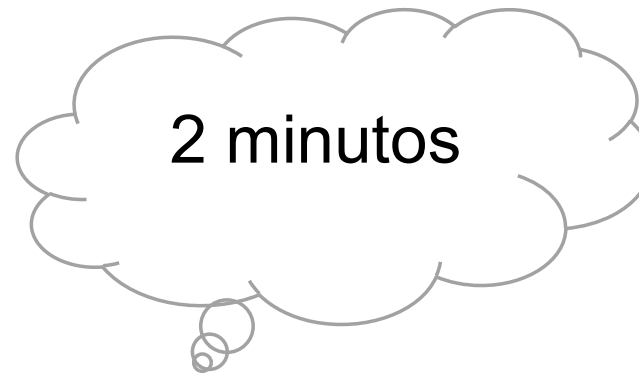
$\langle \rangle$ ,  $\langle -2 \rangle$ ,  $\langle -2, 4 \rangle$ ,  $\langle -2, 4, -3 \rangle$ ,  $\langle -2, 4, -3, 5 \rangle$ ,  $\langle 4 \rangle$ ,  $\langle 4, -3 \rangle$ ,  $\langle 4, -3, 5 \rangle$ ,  $\langle -3 \rangle$ ,  $\langle -3, 5 \rangle$ ,  $\langle 5 \rangle$

**La sub-secuencia con la mayor suma de sus elementos es  $\langle 4, -3, 5 \rangle$  cuya suma es 6**

Si todos los números de la secuencia fueran negativos la sub-secuencia de mayor suma sería la secuencia vacía  $< >$

Note que podemos hacer un algoritmo no recursivo para resolver este problema:

**¿Cómo sería?**



**Hacerlo en casa...  
se demuestra que es menos eficiente  
que el recursivo**

Más formalmente (formulación matemática):

Dada la secuencia de números (algunos pueden ser negativos)  $\langle A_1, A_2, A_3, \dots, A_N \rangle$   
encontrar el máximo valor  $\sum_{k=i}^j A_k$ , para  $i, j$  entre 1 y  $N$ , si  $j < i$  se considera la  
suma igual a 0.

¿Han visto sumatorias?



Veamos como aplicar divide y vencerás (recursión) con la siguiente secuencia:

**ini = 0**

**fin = 7**

4	-3	5	-2	-1	2	6	-2
---	----	---	----	----	---	---	----

Picamos por la mitad (puede ser  $(ini+fin)/2$ ):

**Sub-s 1**

4	-3	5	-2
---	----	---	----

**Sub-s 2**

-1	2	6	-2
----	---	---	----

La subsecuencia de suma máxima :

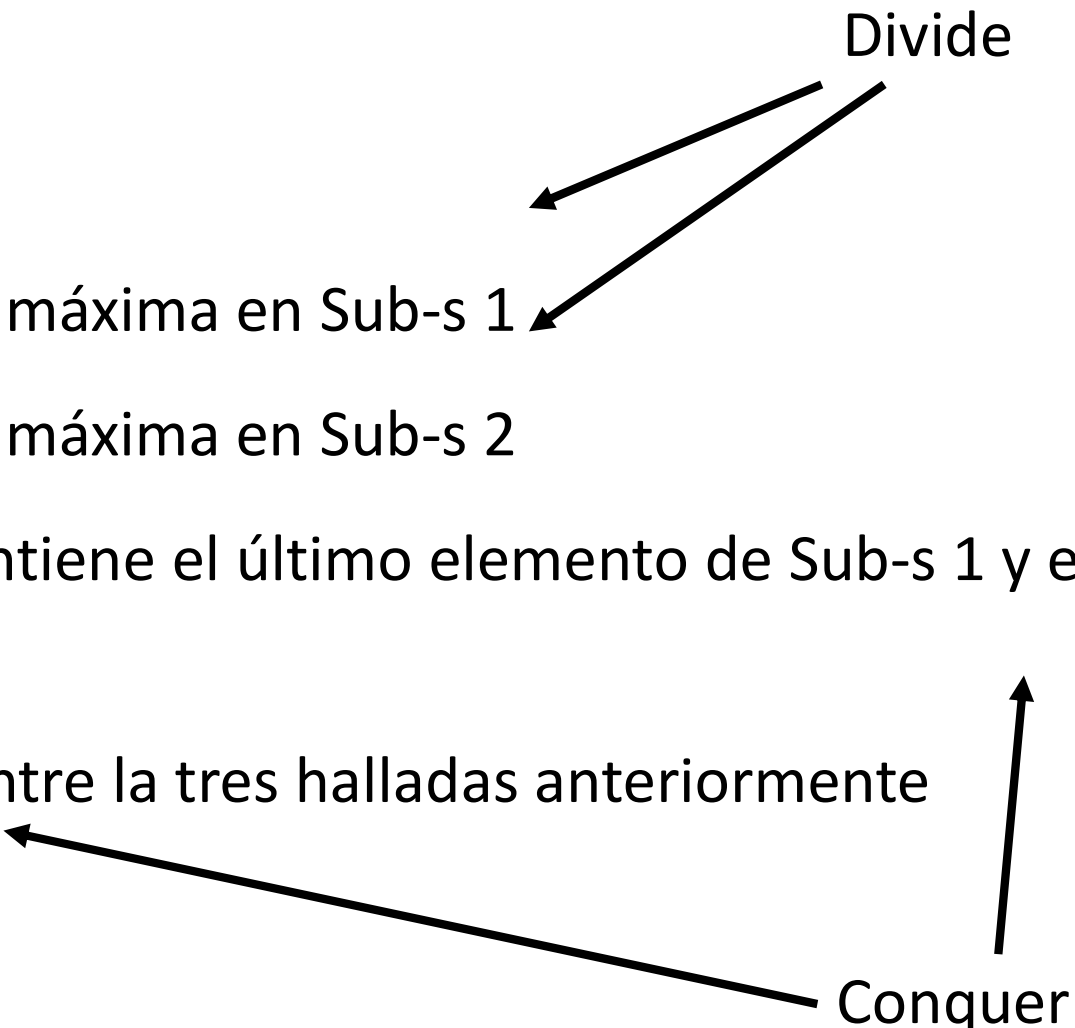
**Caso 1:** se encuentra completamente en la Sub-s 1

**Caso 2:** se encuentra completamente en la Sub-s 2

**Caso 3:** ¿Que faltaría para considerar todas las subsecuencias?....



Por lo tanto podemos resolver el problema por DIVIDE-AND-CONQUER:

- recursivamente buscar la subsecuencia máxima en Sub-s 1
  - recursivamente buscar la subsecuencia máxima en Sub-s 2
  - Buscar la subsecuencia máxima que contiene el último elemento de Sub-s 1 y el primer elemento de Sub-s 2
  - La secuencia buscada será la máxima entre la tres halladas anteriormente
- 

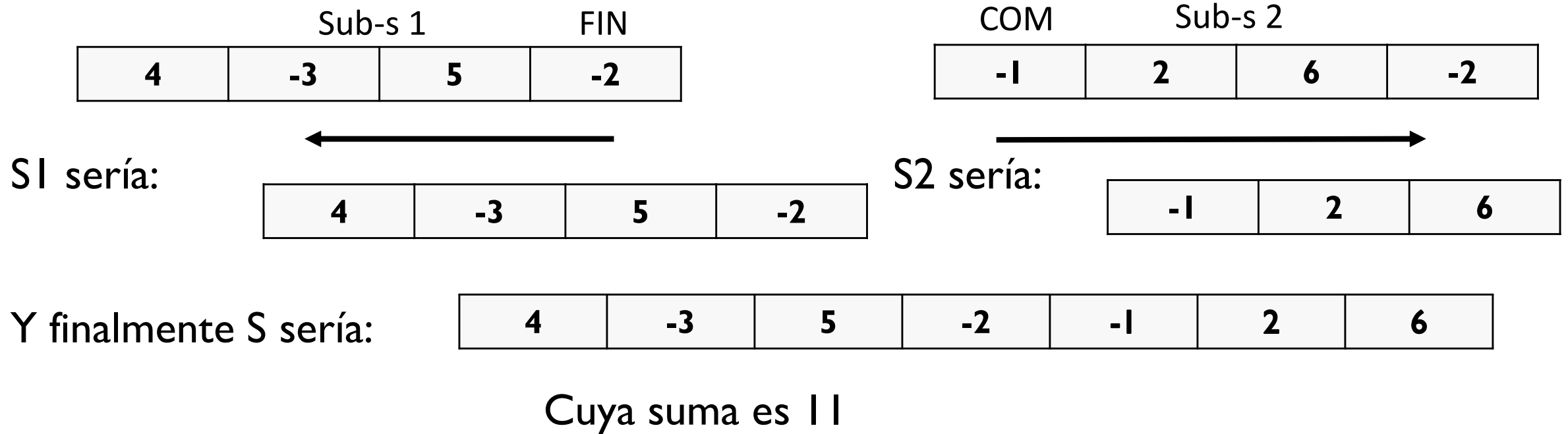
¿Cómo hallamos la subsecuencia máxima  $S$  que contiene el último elemento de Sub-s 1 y el primer elemento de Sub-s 2?

Sean FIN y COM las posiciones de los elementos al final y al comienzo de Sub-s 1 y Sub-s 2 respectivamente

Notemos que para hallar la subsecuencia máxima  $S$  basta con hallar la máxima subsecuencia  $S1$  en Sub-s 1 que termina en FIN, hallar la máxima subsecuencia  $S2$  en Sub-s 2 que comienza en COM y concatenarlas,  $S = S1 || S2$

Para hallar S1 partimos de FIN y vamos hacia la izquierda calculando la máxima subsecuencia.

Para hallar S2 partimos de COM y vamos hacia la derecha calculando la máxima subsecuencia.



### Ejercicio:

Hacer en C++ el programa recursivo para encontrar en una secuencia de números una sub-secuencia contigua con la suma máxima (QUE NO SOLO CALCULE EL MAXIMO SINO QUE CALCULE LA SUBSECUENCIA MAXIMA: INDICE INICIAL Y FINAL DE LA SUBSECUENCIA).

VER PROGRAMA MAXSUM.CPP EN CONTENIDOS DE CLASE 4 DONDE SE DAN 4 ALGORITMOS PARA DETERMINAR LA SUBSECUENCIA MAXIMA (EL RECURSIVO NO DA LA SUBSECUENCIA).

IMPORTANTE: ESTUDIE EN ESE CODIGO. CÓMO HACEN LA COMPARACION DE LOS 4 ALGORITMOS

# **Punteros o Apuntadores (Pointers) en C++**

**C++ permite manejar las direcciones donde se alojan los datos en la memoria del computador:**

**Si x es una variable de cualquier tipo, la expresión:**

**& x**

**Representa la dirección en memoria de x.**

**Y para declarar a una variable que contenga la dirección de memoria de un objeto de un tipo dado, por ejemplo entero, se utiliza:**

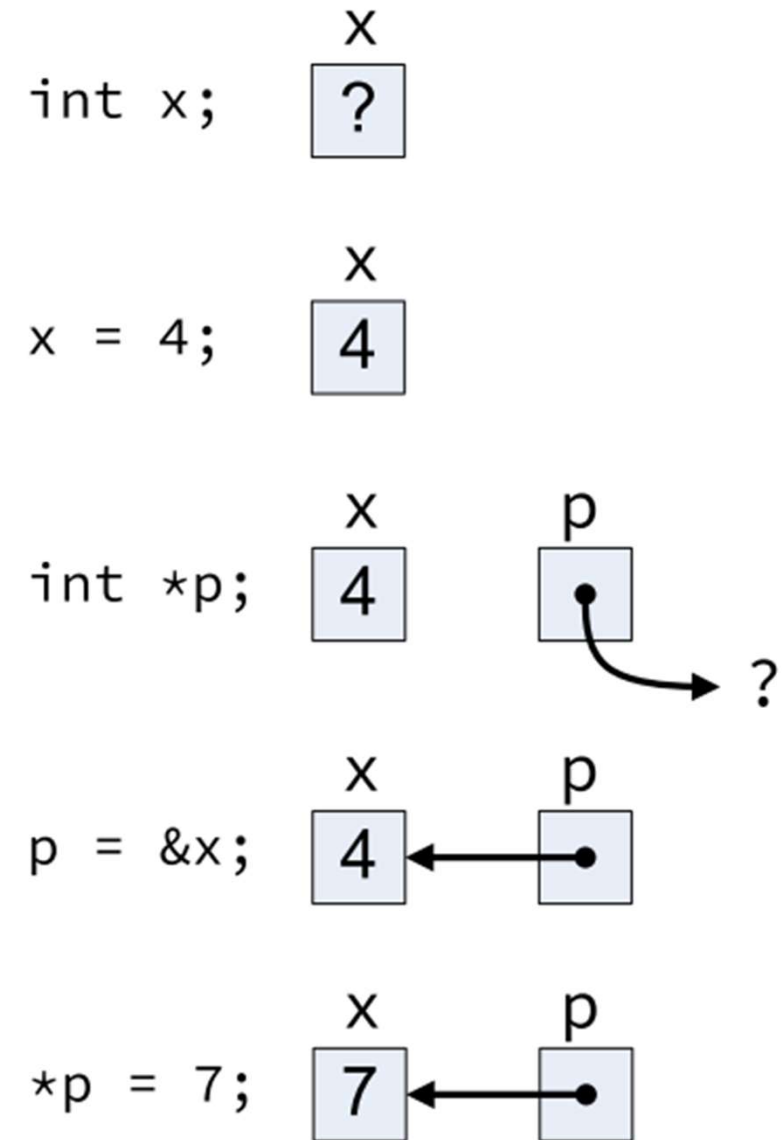
**int \* p;**

**p puede contener la dirección de memoria de una variable tipo entero**

## Ejemplo:

```
int x;
x = 4;
int *p;
p = &x; // p contendrá la dirección de x

*p = 7; // cuando * no se usa en una
        // declaración de un apuntador, la
        // expresión *p es un "pointer
        // dereferencing operator"
        // En este caso copia 7 en la dirección
        // indicada por p. Modifica el valor
        // de la variable x.
```





Existe una palabra reservada de C++:

**nullptr**

Que representa una dirección “nula” . Apunta a nada.

Un apuntador con este valor es un apuntador que no apunta a nada.

**int \*p = nullptr;** // inicializa p con nullptr

**nullptr** puede ser reemplazado por el número 0:

**int \*p = 0;**

Hagamos juntos el siguiente ejercicio:  
Ilustrar gráficamente cada instrucción y lo que imprimirá.

```
int x, y, *p, *q;
p = &x;
q = &y;
x = 100;
y = 200;
*q = *p;
std::cout << x << ' ' << y << '\n';
std::cout << *p << ' ' << *q << '\n';
```

```
int x, y, *p, *q;  
p = &x;  
q = &y;  
x = 100;  
y = 200;  
*q = *p;  
std::cout << x << ' ' << y << '\n';  
std::cout << *p << ' ' << *q << '\n';
```

Ilustrar gráficamente y decir qué imprime este fragmento de código:

```
int x, y, *p, *q;  
x = 5;  
y = 10;  
p = q = &x;  
*p = *q = y;  
std::cout << x << ' ' << y << '\n';
```

```
int x, y, *p, *q;  
x = 5;  
y = 10;  
p = q = &x;  
*p = *q = y;  
std::cout << x << ' ' << y << '\n';
```

Cuidado con esto, podemos tener:

```
int valor = 5; int xxx = 6;
```

```
const int* ptr = &valor; // apuntador a una constante entera
```

```
// Es válido lo siguiente:
```

```
ptr = &xxx; // cambiar a donde el apuntador apunta
```

```
// No es válido:
```

```
*ptr = 10; //no se puede modificar el valor de lo que apunta p
```

```
// es válido:
```

```
valor = 10; // es valido , una forma de cambiar valor a lo que apunta ptr
```

Cuidado con esto, podemos tener:

```
int valor = 5;
```

```
int* const ptr = &valor; //ptr es una constante que apunta a  
// un entero
```

```
// Inválido:
```

```
ptr = nullptr; // No podemos cambiar el valor de un puntero constante
```

```
// Válido:
```

```
*ptr = 10; // Podemos modificar el valor al que apunta el puntero
```

Cuidado con esto, podemos tener:

```
int valor = 5;
```

```
const int* const ptr2 = &valor;
```

```
// Inválido:
```

```
ptr2 = nullptr; // no podemos cambiar el valor del apuntador
```

```
//Inválido:
```

```
*ptr2 = 10;      // no podemos modificar a donde apunta
```



## Podemos crear nuevos objetos utilizando apuntadores

Ejemplo:

```
int *p = new int; // crea un lugar de memoria para  
                // almacenar un int y p lo hace apuntar  
                // a la dirección de memoria creada para un int
```

```
(*p) = 4;
```

```
std::cout << (*p) << "\n";
```

Imprime 4

Todo objeto que se cree con **new** es responsabilidad del programador **liberar** ese espacio creado, porque C++ no lo hará por nosotros.

**C++ solo libera el espacio de las variables locales de** cada función cuando esta termina, pero no libera espacio de objetos creados con **new**

Ejemplo:

```
int *p = new int;
(*p) = 4;
std::cout << (*p) << "\n";
```

**delete p; // dice a C++ que disponga del espacio apuntado por p**

**Variables referencia:** un sobrenombre (alias) para una variable.

```
int x = 5;
```

```
int & r = x; // r es un sobrenombre (alias) de x
```

```
std::cout << "r = " << r << '\n';
```

Imprime r = 5

Decimos que **r** es una **variable tipo referencia** y no es más que otro nombre para **x** (un “alias”). Representan el mismo lugar de memoria.

## Por lo tanto, si tenemos la siguiente declaración:

```
int num, x = 3, y = 10;
int * ptr = &x;
int & ref = y;
int *ptr_otro;
ptr_otro = &ref;
const int & ref1 = x; // ref1 es una constante entera, no podemos modificarla
                      // aunque hace referencia a x que si podemos modificar!!!
                      // int & const ref    da error
```

Podemos escribir:

```
num = ref; // colocar el valor de y en num
num = *ptr; // coloca el valor de x en num
num = *ptr_otro; // coloca el valor de y en num
```

## Importante:

- Al declarar una variable referencia, esta **debe ser inicializada siempre** con una variable `int & ref = y;`
- Una variable referencia **siempre se referirá al mismo lugar de memoria** de cuando se inicializó. Por ejemplo:

```
int x = 5, y = 7;
```

```
int *p = &x;           // p apunta a x
```

```
int& r = x;            // r es otro nombre para x
```

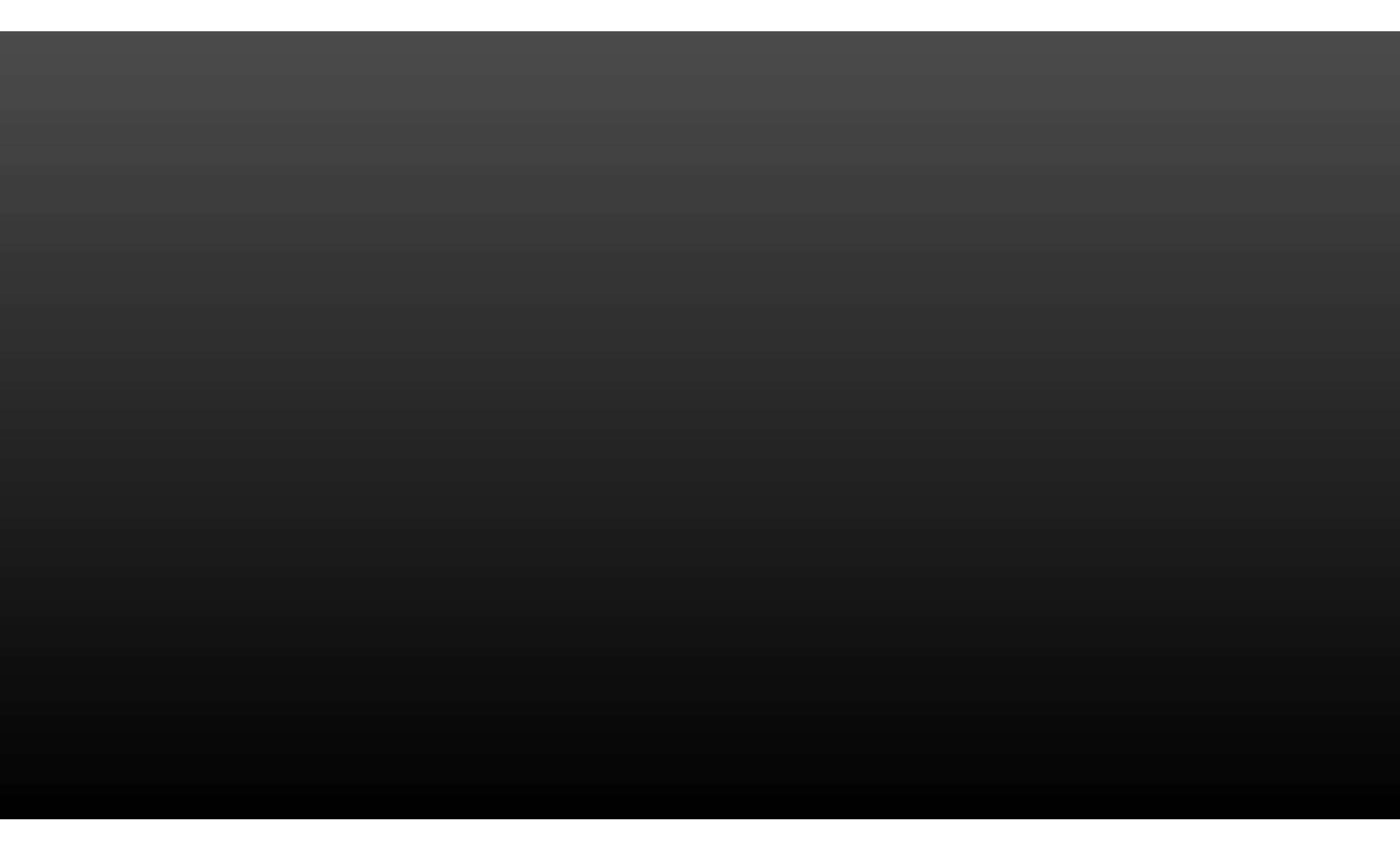
```
p = &y;                // p ahora apunta a y
```

```
r = y;                // a x le asignamos el valor de y
```

## Ilustrar gráficamente y decir qué imprime el siguiente Código:

```
int x = 8, y = 7;  
int *p = &x;  
int& r = y;  
p = &y;  
x = r;
```

```
std::cout << x << ' ' << y << ' ' << (*p) << ' ' << r << '\n';
```



## Paso de parámetros por referencia:

C++ permite pasar a una función una variable **por referencia**, es decir, **el objeto (o variable) que se pasa como parámetro real en una llamada será el mismo objeto que recibe el parámetro formal de la función.**

Por lo tanto **la función podrá modificar el valor del objeto pasado por referencia**, a diferencia del paso por valor que hace una copia del valor del objeto.

Hay diferentes formas de pasar una variable por referencia, dependiendo de si usamos apuntadores o variables referencia.

Veamos...



## Hagamos una función que intercambia los valores de dos variables utilizando apuntadores:

```
#include <iostream>
/*
 * intercambio(a, b)
 * Intercambia los valores de memoria de los
 * objetos referenciados a y b
 * Se intercambian los valores de las variables
 * del programa que llama a la función
 */
void intercambio(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main() {
    int var1 = 5, var2 = 19;
    intercambio(&var1, &var2);
    std::cout << "var1 = " << var1
                << ", var2 = " << var2 << '\n';
}
```

Imprime: var1 = 19, var2 = 5

## Hagamos una función que intercambia los valores de dos variables utilizando variables referencia:

```
#include <iostream>
```

```
/*
```

```
* intercambio(a, b)
```

```
* Intercambia los valores de memoria de los
```

```
* objetos referenciados por a y b
```

```
* Se intercambian los valores de las variables
```

```
* del programa que llama a la función
```

```
*/
```

```
void intercambio(int& a, int& b) {
```

```
    // a y b son referencias a
```

```
    // los parámetros reales
```

```
    int temp = a;
```

```
    a = b;
```

```
    b = temp;
```

```
}
```

```
int main() {
```

```
    int var1 = 5, var2 = 19;
```

```
    intercambio(var1, var2); // los parámetros
```

```
        // reales son las propias variables
```

```
    std::cout << "var1 = " << var1
```

```
        << ", var2 = " << var2 << '\n';
```

```
}
```

Imprime: var1 = 19, var2 = 5

## Observaciones:

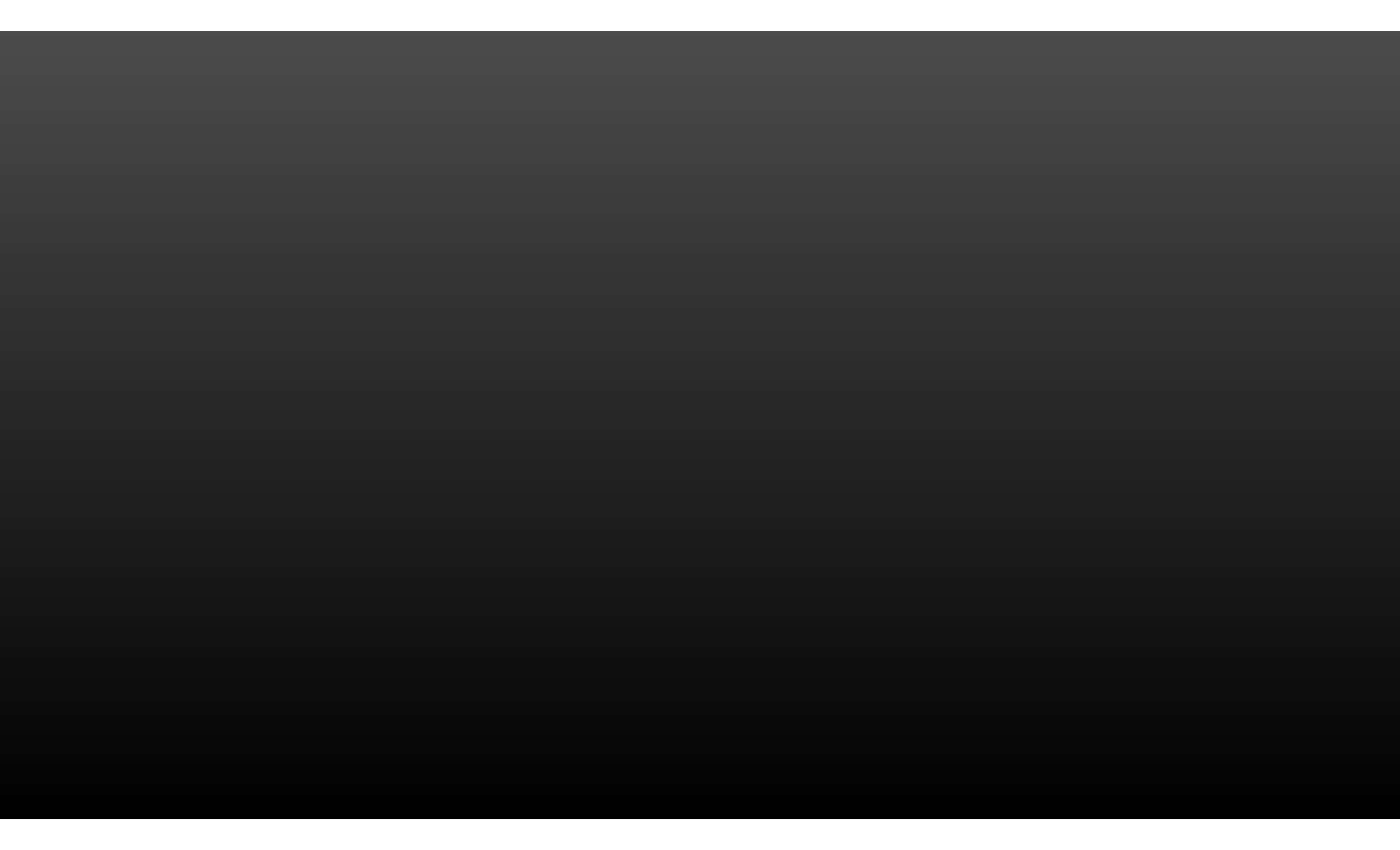
- Usando **variables referencia**, el paso de parámetros desde la función que llama **se ve igual a un paso por valor**.
- **Hay que tener cuidado** con las funciones que tienen parámetros pasados por referencia (sea con apuntadores o variables referencia) porque pueden crear lo que se conoce como “**efectos colaterales**” (side effects), cambios no deseados.
- El desarrollo de programas es “más limpio” cuando las funciones pueden tratarse como **cajas negras** que realizan cálculos de forma aislada. **Pero a veces no es posible y hay que estar consciente.**

**Podemos pasar por referencia un vector (o un tipo struct) a una función:**

```
void f(vector<string>& v){
    .....
}
```

```
int main( ){
    vector<string> st {"aa","bb"};
    f(st);
}
```

**Hagamos juntos una función llamada “limpiar” que inicialice todos los elementos en cero de un vector de enteros**



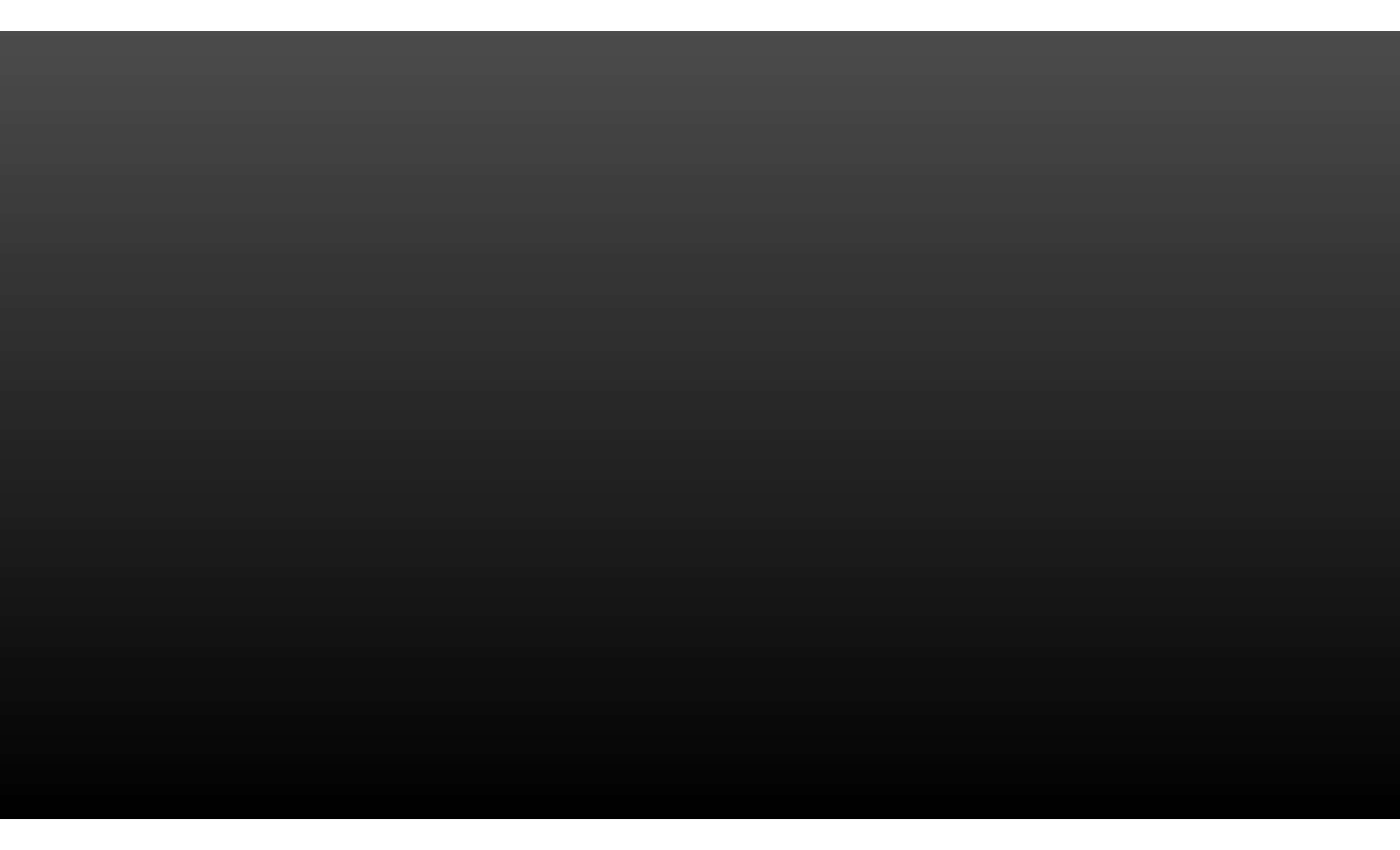
```
#include <vector>
using std::vector;

void limpiar(vector<int>& v){
    for (int i=0; i<v.size() ; i++)
        v[i] = 0;
}

int main( ){
    <int> vec_ent {4, 44 , 56, 3, 2};
    limpiar(vec_ent);
}
```

¿Cómo sería “limpiar” usando apuntadores?





```
#include <vector>
using std::vector;

void limpiar(vector<int>* v){
    for (int i=0; i<(*v).size() ; i++)
        (*v)[i] = 0;
}

int main( ){
    <int> vec_ent {4, 44 , 56, 3, 2};
    limpiar(&vec_ent);
}
```

**Podemos pasar variables por referencia a una función y que no podamos modificarla dentro de la función. Veamos esta función que imprime un vector de enteros:**

```
void imprimir_vector(const std::vector<int>& vec) {
    std::cout << "{";
    int len = vec.size();
    if (len > 0) {
        for (int i = 0; i < len - 1; i++)
            // coma después de cada elemento
            std::cout << vec[i] << ",";
        // sin coma último elemento
        std::cout << vec[len - 1];
    }
    std::cout << "}\n";
}
```

Garantizamos que **vec** no será modificado dentro de la función.

**vec** es una referencia a un vector constante, es decir, que no podemos modificar en `print_vector`

Volvamos al problema: Hacer una función recursiva que imprima un vector de enteros. Pero ahora utilizando el paso de parámetros por referencia **usando variables referencia** (más eficiente porque no crea nuevos vectores.....)

Veamos.....

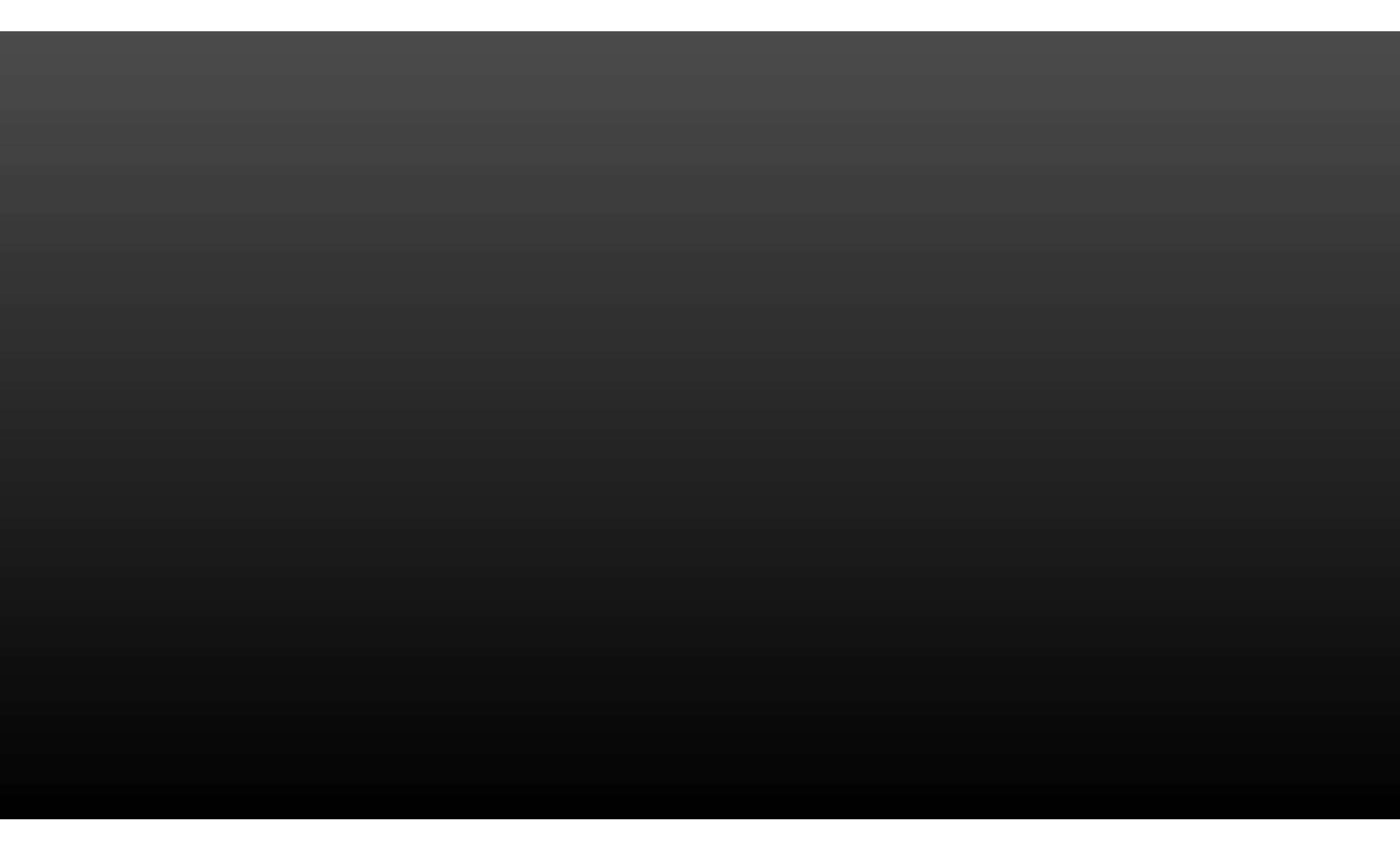
#include <iostream>	void imprimir ( <b>const vector&lt;int&gt;&amp; v</b> ) {
#include <vector>	imprimir_rec(0, v);
using namespace std;	std::cout << '\n';
void imprimir_rec (int i, <b>const vector&lt;int&gt;&amp; v</b> ) {	}
if (i > (v.size()-1)) return;	int main( ){
std::cout << v[i] << " " ;	vector<int> v {1,2,3,4};
imprimir_rec (i+1, v) ;	imprimir(v);
}	}

Siempre será el mismo vector

Como **v** no se modifica se pasa por referencia pero como constante (**const**)

Volvamos al problema: Hacer una función recursiva que imprima un vector de enteros. Pero ahora utilizando el paso de parámetros por referencia **usando apuntadores** (igual de eficiente que antes pero más confusa la notación)

Pensémoslo juntos.....



```
#include <iostream>

#include <vector>

using namespace std;

void imprimir_rec (int i, const vector<int> * v) {
    if (i > ((*v).size()-1)) return;
    std::cout << (*v)[i] << " ";
    imprimir_rec (i+1, v);
}
```

Garantiza que NO se modifica

Siempre será el mismo vector

```
void imprimir (const vector<int> * a_v) {
    int i = 0;
    imprimir_rec(i, a_v);
    std::cout << '\n';
}

int main( ) {
    vector<int> v {1,2,3,4};
    imprimir(&v);
}
```

Como **v** no se modifica se pasa como por referencia pero constante



Como ya aprendimos el paso de parámetro por referencia, podemos modificar **el tipo Conjunto (en forma modular con .h y .cpp)** que habíamos hecho con solo paso por valor, lo cual era ineficiente y teníamos que hacer cosas como:

**conj = insertar(conj, i);**

Ahora podemos insertar un elemento a un conjunto **utilizando paso de parámetros por referencia.**

Veamos....

## El archivo conjunto\_ref.h:

```
#include <vector>

using namespace std;

//typedef vector<int> Conjunto; es igual a using
using Conjunto = vector<int> ;

void insertar(Conjunto & , int); // note que el parámetro es
                                // por referencia

void imprimir(const Conjunto & ) ;

bool pertenece(const Conjunto & , int );
```

## El archivo conjunto\_ref.cpp:

```
#include <iostream>
#include "conjunto_ref.h"

void insertar(Conjunto &v, int i){
    // Verificar que no esté para
    // poderlo insertar
    for (int j=0; j<v.size(); j++){
        if (i==v[j]) return;
    }
    v.push_back(i);
}
```

```
bool pertenece(const Conjunto &v, int i){
    // Devuelve true sii i esta en v
    for (int j=0; j<v.size(); j++){
        if (i==v[j]) return true;
    }
    return false;
}

void imprimir(const Conjunto &conj){
    for (int j=0; j<conj.size(); j++)
        std::cout << conj[j] << " ";
}
```

## El archivo de prueba prueba.cpp: Note que **NO** sabemos como se ha implementado el tipo **Conjunto**

```
#include <iostream>
#include "conjunto_ref.h"
int main(){
    Conjunto conj; //crea un conjunto vacío
    insertar(conj,0); // se modifica la variable conj
    insertar(conj,1);
    insertar(conj,2);
    imprimir(conj);
    int i=3;
    std::cout << "¿Está " << i << " en el conjunto?: "
        << std::boolalpha << pertenece(conj,i);
}
```

**En el laboratorio del lunes próximo  
utilizaremos este código** que implementa el TDA  
Conjunto y alguna de sus operaciones, para que lo  
**tengan a la mano (estará en CONTENIDOS  
en Módulo 7)**

## Funciones de orden superior: Podemos pasar una función como un parámetro de la función, pasando un apuntador a la función.

- Ejemplo:

```
#include <iostream>
int sumar(int x, int y) {
    return x + y;
}
int multiplicar(int x, int y) {
    return x * y;
}
int evaluar(int (*f)(int, int), int x, int y) {
    return f(x, y);
}
```

```
int main() {
    std::cout << sumar(2, 3) << '\n';
    std::cout << multiplicar(2, 3) << '\n';
    std::cout << evaluate(&sumar, 2, 3) << '\n';
    std::cout << evaluate(&multiplicar, 2, 3) << '\n';
}
```

**Ejercicio para la casa:**

**Hacer una función que ordene un vector de enteros. Se le pasa la función que permite ordenar (ejemplo: ascendentemente, descendientemente)**

**¿Saben ordenar un vector?**

**Hablemos un poco de arreglos (que usaré poco):**

**Los arreglos son como los vectores y es un tipo estructurado primitivo de C.**

- Podemos declarar un arreglo de la siguiente forma:

**`int list[25];`**

- **Se DEBE dar el tamaño en la declaración mediante un entero (25) o una constante.**  
No se puede modificar el tamaño durante todo el programa.
- La declaración **`int arreglo[0];`** es ilegal pues un arreglo **NO puede tener tamaño 0**
- Para determinar el número de elementos de un arreglo:

**`sizeof(list) / sizeof(list[0]).`**



## Ejemplos:

```
int x;
```

```
std::cin >> x;
```

```
int list_1[x];
```

```
std::vector<int> list_2(x);
```

**ILEGAL.** El tamaño  
debe ser constante

Legal para un vector

```
double collection[] = { 1.0, 3.5, 0.5, 7.2 };
```

ó 

```
double collection[] { 1.0, 3.5, 0.5, 7.2 };
```

**Legal:** crea un arreglo de 4 elementos tipo double.  
Su tamaño será 4 durante todo el programa

**Podemos pasar un arreglo como parámetro, pero en realidad se pasa “por referencia”**

```
#include <iostream>
void imprimir(const int b[], int n)
{ // todo el arreglo es constante
    for (int i = 0; i < n; i++)
        std::cout << b[i] << " ";
    std::cout << '\n';
}
void borrar(int a[], int n) {
    for (int i = 0; i < n; i++)
        a[i] = 0;
}
```

```
int main() {
    int list [ ] = { 2, 4, 6, 8 };
    imprimir(list, 4);
    borrar(list, 4);
    imprimir(list, 4);
}
```

**El nombre de un arreglo se comporta como un apuntador al primer elemento del arreglo.**

Cuando **list** es pasado a las funciones **imprimir( )** y **borrar( )**, en realidad **se pasa la dirección del primer elemento del arreglo, SE DEBE PASAR EL TAMAÑO**

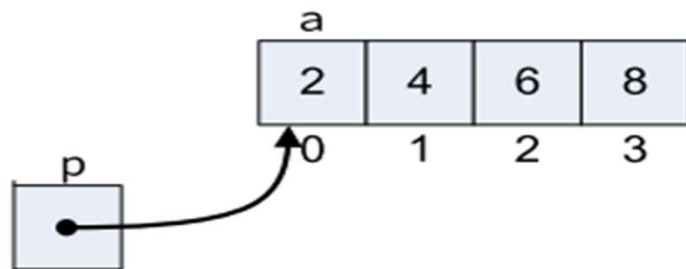
Para ilustrar que el nombre de un arreglo es en realidad un  
apuntador al primero.

```
int a[] = { 2, 4, 6, 8 }, *p
```

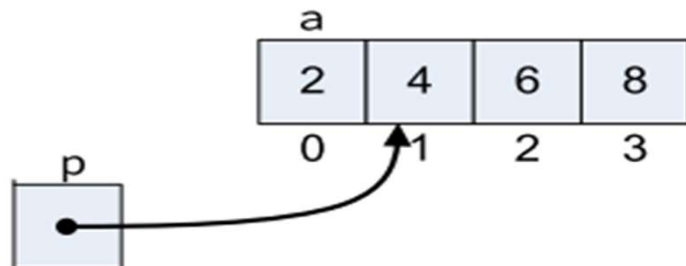


apuntador a entero

```
p = a;
```



```
p++;
```



Incluso **p[2]** es **válido** e indica el valor (6)  
del elemento en la posición 2 del arreglo  
original **a**

Si **p = a** entonces **p++** apunta al elemento  
en la posición 1 de **a**. Es decir **\*(p++)** es  
igual a 4

**No podemos copiar un arreglo en otro:**

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

```
b = a; // ERROR
```

**Con vectores podemos:**

```
vector<int> a(10, 88);  
vector<int> b;  
b = a; // crear una copia de a y la coloca en b
```

## Arreglos dinámicos:

Podemos crear **arreglos dinámicos**, es decir que podemos definir su tamaño en tiempo de ejecución (y no de compilación):

```
int x = 3;
```

```
int *p = new int[ x+3 ]; // note que el tamaño se determina en  
                        // tiempo de ejecución y no de compilación  
                        // aunque permanecerá con ese tamaño el resto  
                        // de la ejecución del programa
```

**p** es un apuntador a un arreglo de 6 elementos.

Para liberar el espacio de este arreglo usamos: **delete [ ] p;**

**Ejemplo de uso de arreglos dinámicos:**

**Un programa que hace el promedio de varios números introducidos por teclado.**

**Se pide inicialmente la cantidad de números a promediar.**

```
#include <iostream>
int main() {
    double suma = 0.0;
    double *numeros; // numeros es un apuntador, no un arreglo
    int tam; // Cantidad de números a leer

    // Leer la cantidad de números a procesar
    std::cout << "Coloque la cantidad positiva de números a procesar: ";
    std::cin >> tam;

    if (tam > 0) { // verifica que sea positivo
```

```
std::cout << "Escriba " << tam << " numeros: ";
// Reservar la cantidad exacta requerida
numeros = new double[tam]; // arreglo dinámico, numeros es apuntador
// El usuario escribe por teclado los números
for (int i = 0; i < tam; i++) {
    std::cin >> numeros[i];
    suma += numeros[i];
}
std::cout << "El promedio de los números ";
for (int i = 0; i < tam - 1; i++)
    std::cout << numeros[i] << ", ";
// sin coma al final, el último número escrito
std::cout << numeros[tam - 1] << " es " << suma/tam << '\n';
delete [] numeros; // liberar el arreglo
```



**Hagamos juntos un programa que cree un arreglo de enteros y una función a la que se le pase el arreglo y devuelva un arreglo con solo los elementos en posiciones impares.**

**Devolvemos el nombre del arreglo que sabemos es un apuntador al primer elemento del arreglo.**

**El prototipo de la función es:**

**`int * impares(int a [ ]; int n);`**



```
#include <iostream>

int * impares(int a [], int n){
    int * b = new int[n/2];
    for (int i=0; i< (n/2) ; i++)
        b[i]= a[i*2+1];
    return b;
}

void imprimir(const int b[], int n) {
    for (int i = 0; i < n; i++)
        std::cout << b[i] << " ";
    std::cout << '\n';
}
```

```
int main( ){
    int list1[] = { 2, 4, 6, 8 };
    int *p = impares(list1,4);
    imprimir(p,2);
    delete [] p;
    return 0;
}
```

## PREGUNTAS???

