

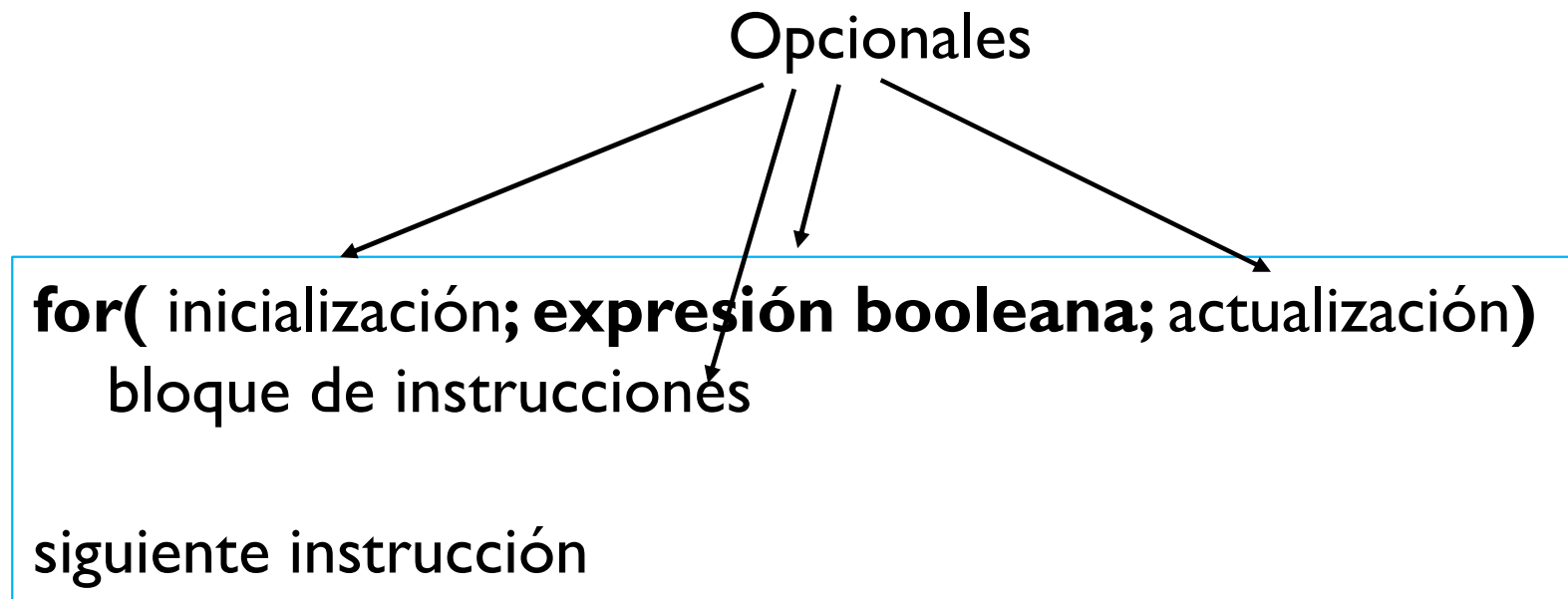
Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

Continuamos con funciones en C++

La instrucción **for**:



Ejemplos:

```
// imprimir todos los pares (j,i) tales que j <= i <= 10  
  
for( int i = 1; i <= 10; i++ ) // alcance de i solo  
                                // en el bloque del for  
    for( int j = 1; j <= i; j++ )  
        std::out << " ( " << j << " , " << i << " ) "  
        << std::endl;
```

Otros ejemplos:

```
int i = 0;  
for( ; i <= 5; )  
    std::cout << i++ << '\n';
```

```
int i=0,sum=0, n=10;  
for( ; i <= n; i++)  
    sum += i;
```

```
int i=0,sum=0, n=10;  
for( ; i <= n; i++, sum+=i )  
    { };
```

Discutamos juntos como haríamos un programa en C++ que lea un número entero no negativo N y luego imprima todos los números primos menores o iguales a N . (Recuerde que un número primo es un entero mayor que 1 divisible solo por el mismo y por 1)

Para probar que un número M es primo basta probar que no es divisible por los números enteros que van desde 2 hasta la raíz cuadrada de M . ¿Por qué hasta raíz?

Respuesta: Si M admite un factor ≥ 2 (no es primo), es decir $M = A \times B$ y $A \leq B$ entonces $A \leq R$ (la raíz de M)

Porque si A fuese mayor que R , como $B \geq A$ entonces $A \times B$ sería mayor que $R \times R$ que es igual a M , una contradicción porque $A \times B$ es igual a M

Por lo tanto basta probar que M no tiene factores entre 2 y R

Podemos usar `sqrt(x)` para determinar la raíz cuadrada. Está en la biblioteca `<cmath>`

Recordemos: Imprimir los números primos hasta un entero leído por pantalla

```
#include <iostream>
#include <cmath>
```

```
int main() {
    int N;
    double raiz_de_valor;
    std::cout << "Imprimir primos hasta qué valor? ";
    std::cin >> N;
    for (int valor = 2; valor <= N; valor++) {
        // se comprueba si valor es primo
        bool es_primo = true; /* Inicialmente suponemos
                               que es primo para entrar al for e ir verificando */
```

```
        raiz_de_valor = sqrt(valor); //usamos la función sqrt( )
        for (int factor_de_prueba = 2;
             es_primo && (factor_de_prueba <= raiz_de_valor);
             factor_de_prueba++)
            es_primo = (valor % factor_de_prueba != 0);
        if (es_primo)
            std::cout << valor << " "; // Imprime un primo
    }
    std::cout << '\n'; // Ir a la siguiente línea
    return 0;
```

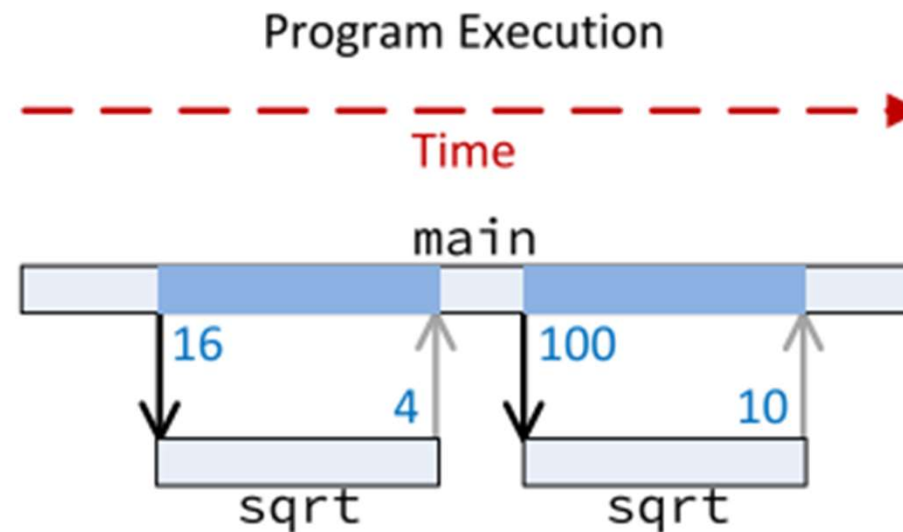
Vemos que podíamos utilizar una **función** de la biblioteca “**cmath**” de C++ para determinar la raíz cuadrada de un número: `sqrt ()`.
¿Cómo funciona una llamada a una función?

```
int main() {
    double value;

    // Assign variable
    value = 16;

    // Compute s square root
    double root = sqrt(value);

    // Compute another
    root = sqrt(100);
}
```



parámetro

Desde el punto de vista del usuario, al utilizar una función el usuario necesita conocer:

- Su nombre: **sqrt**
- Los tipos de los parámetros de la función: **double**
- El tipo del resultado de la función: **double**

Esta información constituye **el prototipo de la función**:

double sqrt (double)

La función **sqrt()** viene en tres formas:

- `double sqrt(double)`
- `float sqrt(float)`
- `long double sqrt(long double)`

El nombre de la función es la misma pero cambia el tipo de los parámetros. Son tres versiones distintas de la misma función pero para parámetros de diferente tipo.

Decimos que la función **sqrt()** está “sobrecargada” (**overloaded**). Luego veremos este concepto en más detalle

Hay funciones que no necesitan parámetros o que no devuelven ningún valor de retorno:

void exit (int): detiene la ejecución del programa y devuelve un número al sistema operativo para saber si el programa terminó normalmente o no. La palabra reservada “void” indica que la función no devuelve ningún valor.

int rand (): genera un número entero aleatorio entre 0 y RAND_MAX (una constante)

Las dos están en la biblioteca **<cstdlib>**

Programa que utiliza las funciones `clock()` y `sqrt()` de la biblioteca estándar de C++ para determinar el tiempo que tarda un programa en calcular los primeros 1000 números primos:

```
#include <iostream>
#include <ctime>
#include <cmath>

/* Imprimir los primeros 1000 números primos y al final
   imprimir el tiempo que tardó el programa */

int main() {
    clock_t tiempo_com = clock(), // tiempo de comienzo
    tiempo_fin; // tiempo de finalizacion

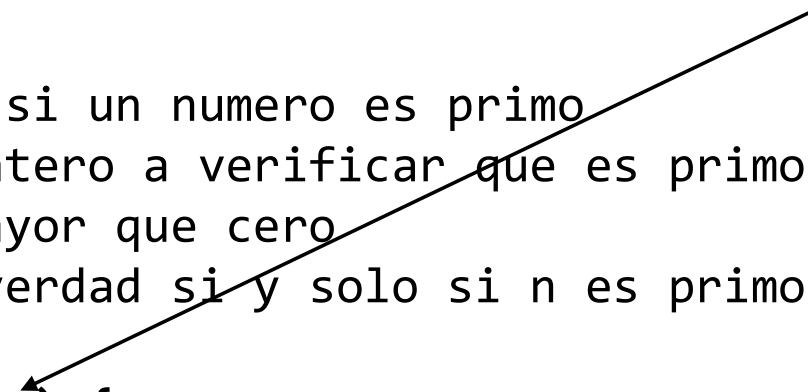
    for (int valor = 2; valor <= 1000; valor++) {
        // ver si "valor" es primo
        bool es_primo = true; // Suponemos que es primo
        // probar todos los factores de 2 a sqrt(valor)
```

```
        for (int factor_prueba = 2;
             es_primo && factor_prueba <= sqrt(valor);
             factor_prueba++)
            es_primo = (valor % factor_prueba != 0);
        if (es_primo)
            std::cout << valor << " "; // Imprimir primo
    }
    std::cout << '\n'; // ir a la linea siguiente
    tiempo_fin = clock();
    // Imprimir tiempo transcurrido
    std::cout << "Tiempo transcurrido en segundos: "
              << static_cast<double>(tiempo_fin -
                                    tiempo_com)/CLOCKS_PER_SEC
              << " seg." << '\n';
}
```

Podemos escribir nuestras propias funciones en C++. Escribamos otra vez el programa anterior para determinar el tiempo que tarda un programa en calcular los primeros 1000 números primos:

```
#include <iostream>
#include <ctime>
#include <cmath>      // necesaria por sqrt
/*
 *  es_primo(n)
 *      Determina si un numero es primo
 *      n es el entero a verificar que es primo
 *      n deber mayor que cero
 *      Devuelve verdad si y solo si n es primo
 */
bool es_primo(int n) {
    bool result = true; // De antemano lo suponemos primo
    double  raiz = sqrt(n);
    // Probamos con todos los posibles factores de 2 a raíz de n
    for (int posible_factor = 2; result && posible_factor <= raiz;
        posible_factor++)
        result = (n % posible_factor != 0);
    return result;
}
```

Parámetros formales (uno en este caso)



```

/* Imprimir los primeros 1000 números primos y al final
   imprimir el tiempo que tardó el programa */
int main() {

    clock_t tiempo_com = clock(), // tiempo de comienzo
            tiempo_fin; // tiempo de finalizacion

    for (int valor = 2; valor <= 1000; valor++)
        // ver si "valor" es primo
        if (es_primo(valor))
            std::cout << valor << " "; // Imprimir primo

    std::cout << '\n'; // ir a la linea siguiente
    tiempo_fin = clock();
    // Imprimir tiempo transcurrido
    std::cout << "Tiempo transcurrido en segundos: "
        << static_cast<double>(tiempo_fin - tiempo_com)/CLOCKS_PER_SEC
        << " seg." << '\n';
}

```

**Parámetros reales
(uno en este caso)**

Otro ejemplo: Escribamos una función que calcula la raíz cuadrada por un método conocido como el método de Newton

```
#include <cmath>
// Calcula la raiz cuadrada de un numero x
double raiz_cuadrada(double x) {
    double diff;
    // Raiz provisional
    double root = 1.0;
    do { // Iterar hasta conseguir una
        // aproximacion de 4 decimales
        root = (root + x/root) / 2.0;
        // Determinar la aproximacion
        diff = root * root - x;
    } while (diff > 0.0001 || diff < -0.0001);
    return root;
}
```

diff:
variable
local a la
función

Valor absoluto de diff
mayor que 0.0001)

Parámetros formales
(uno en este caso)

```
int main() {
    // Comparar nuestra funcion con la
    // de la biblioteca estandar para los
    // reales entre 1 y 10, con paso de 0.5
    for (double d = 1.0; d <= 10.0; d += 0.5)
        std::cout << std::setw(7)
            << raiz_cuadrada(d)
            << " : " << sqrt(d) << '\n';
}
```

Parámetros reales (uno
en este caso)

Los parámetros reales de una llamada a función pueden ser cualquier expresión cuyo tipo sea el mismo del parámetro formal correspondiente:

Ejemplos:

- `sqrt(2.6 + 4.5)`
- `sqrt (max (x, y))` : suponemos que **max** es una función que devuelve el máximo entre dos números. Las variables `x` e `y` deben ser números
- `sqrt (sqrt (16))` convierte implícitamente el entero 16 a double

- En el caso de la función **es_primo()** el parámetro se pasa **POR VALOR**, es decir que el **valor** del parámetro real es **COPIADO** al parámetro formal. Si **es_primo()** modificara su parámetro formal en su cuerpo, **NO** se modifica el valor del parámetro real

```
void xx (int x){
    x++;      //x es local a la función  xx ( )
    std::cout << x << std::endl;
}
int main() {
    int x = 2; // x es local a la función main( )
    xx(x);
    std::cout << x << std::endl;
}
```

¿Qué imprimiría?:
3
2

Es importante documentar los programas y en especial las funciones (**Esto será exigido y calificado en los proyectos**). Al comienzo de la definición de una función se debe colocar un comentario que describe la naturaleza de la función:

- Describir qué hace la función, el propósito de la función.
- Describir cada parámetro formal.
- Describir que devuelve la función.

Ejemplo de documentación de una función:

```
/*  
 * distancia(x1, y1, x2, y2)  
 * Descripción: Calcula la distancia entre dos puntos del plano cartesiano  
 * x1 es la coordenada x del primer punto  
 * y1 es la coordenada y del primer punto  
 * x2 es la coordenada x del segundo punto  
 * y2 es la coordenada y del segundo punto  
 * Devuelve la distancia entre (x1,y1) y (x2,y2)  
 */  
double distancia(double x1, double y1, double x2, double y2) { ... }
```

- Las variables declaradas en una función son locales a la función
- Podemos tener **variables y constantes globales**, las cuales se declaran fuera de cualquier función, incluso fuera de **main()**
- Una variable o constante global es accesible a cualquier función. **Esto hace a la función (si la usa) dependiente de un ente externo (crear efectos colaterales)!!**
- Si una variable global tiene igual nombre a una variable local a una función la podemos acceder en la función anteponiendo ::
Ejemplo: x esta declarada en f() y también x es una variable global. Dentro de f() la usamos **::x para diferenciarla de la local**

Una variable estática se define dentro de una función para preservar su valor entre llamadas de la función.

```
#include <iostream>
```

```
/*
 * contar
 * Mantiene un contador estático.
 * Devuelve el valor del contador.
 */
```

```
int contar() {
    // El valor e cnt es mantenido
    // entre llamadas por ser declarado estático
    static int cnt = 0;
    return ++cnt; // incremento y devuelve cnt
}
```

```
int main() {
    // Cuenta hasta 10
    for (int i = 0; i < 10; i++)
        std::cout << contar() << ' ';
    std::cout << '\n';
}
```

Una función puede tener parámetros que se pueden definir por defecto.

- Ejemplo:

```
#include <iostream>
// Imprimir un contador desde n hasta 0
// El valor por defecto es 10
void imprimir_decreciendo(int n=10) {
    while (n >= 0)
        std::cout << n-- << '\n';
}
int main() {
    imprimir_decreciendo(5);
    std::cout << "-----" << '\n';
    imprimir_decreciendo( );
}
```

← Sin parámetro, assume 10

Una función puede ser “sobrecargada” (overloaded): funciones que hacen básicamente lo mismo pero con diferente número de parámetros)

```
void f ( ) { /* ... */ } // esta versión no posee parámetros
```

```
void f (int x) { /* ... */ } // esta versión posee un parámetro entero
```

```
void f (int x, double y) { /* ... */ } // esta versión posee dos parámetros
```

NO podemos tener como sobrecarga de la función anterior:

```
int f (int x) { /* ... */ } // Dos funciones con mismo nombre y mismos parámetros  
// pero que devuelvan distintos tipos pues hay ambigüedad  
// al llamarlas desde otro programa.
```

La FIRMA (signature) de una función consiste en el nombre de la función y su lista de parámetros. Ejemplo: **f (int x)**

No podemos tener dos funciones con la misma firma pero que devuelvan tipos diferentes

El tipo estructurado **arreglo**

C++ posee como todo lenguaje de programación el tipo **arreglo (array)**. Un arreglo es una estructura de datos que permite **implementar** una **secuencia** de elementos del mismo tipo. Cada elemento tiene una posición comenzando desde 0

Por ejemplo un arreglo de enteros:

```
int list[3];  
list[0] = 5;  
list[1] = -3;  
list[2] = 12;
```

list		
5	-3	12
0	1	2

Otros ejemplos:

```
std::char aa[3] {'a','b','c'};  
int bb[3] {2,3,4};
```

El tipo “array” es heredado de C y presenta complicaciones al utilizarlo (luego veremos por qué). Por lo que a lo largo del curso utilizaremos el tipo **vector** (objeto de C++). Que es similar a un arreglo pero tiene mejor comportamiento que los arreglos y tiene muchas más ventajas.

Sin embargo, cuando veamos manejo de memoria dinámica hablaremos de arreglos dinámicos para aprender.

Vectores:

Un vector en C++ permite almacenar una secuencia de objetos del mismo tipo (como los arreglos). Para utilizarlos debemos colocar la directiva:

#include <vector>

Y como forma parte de la biblioteca estándar de C++, habría que declararlos de esta forma:

```
std::vector<int>    vec_a;    // vec_a sería un vector de enteros
```

Si al comienzo de nuestro programa colocamos la directiva: **using std::vector;**
también lo podemos declarar sin el std::

`vector<int> vec_a;`

Formas de declarar un vector:

```
vector<int> vec_a;  
vector<int> vec_b(10);  
vector<int> vec_c(10, 8);  
vector<int> vec_d{ 10, 20, 30, 40 };
```

Podemos acceder a sus elementos como en arreglos:

std::vector<int> lista (3); // Declara **lista** como un vector de 3 enteros

lista[0] = 5 ; // Colocar 5 al primer elemento en posición 0

lista[1] = -3 ; // Colocar -3 al segundo elemento en posición 1

lista[2] = 12 ; // Colocar 12 al ultimo elemento en posición 2

std::cout << lista[2] << '\n'; // Imprimir el elemento en la posición 2

// las posiciones comienzan desde 0

Un **vector** puede contener objetos de cualquier tipo:

```
std::vector<double> reales { 1.0, 3.5, 0.5, 7.2 };
```

```
std::vector<char> letras { 'a', 'b', 'c' };
```

Es válido: referirse al elemento en la posición $x + y$, `letras[$x + y$]`, siempre y cuando **x** e **y** sean tipo entero y sea una posición válida.

El tipo vector en realidad es UNA CLASE (mecanismo que permite definir nuevos objetos en C++) que “enmascara” los arreglos primitivos del lenguaje C.

Ejemplo que calcula el promedio de los elementos de un vector de números:

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

int main() {
    double suma = 0.0;
    const int CANTIDAD_DE_ELEM = 10;
    vector<double> numeros(CANTIDAD_DE_ELEM);
    std::cout << "Por favor coloque "
                << CANTIDAD_DE_ELEM << " numeros "
                << << " en lineas diferentes ";
    // Puede leer enteros o reales
    for (int i = 0; i < CANTIDAD_DE_ELEM; i++) {
        std::cin >> numeros[i];
        suma += numeros[i];
    }

    std::cout << "El promedio de ";
    for (int i = 0; i < CANTIDAD_DE_ELEM - 1; i++)
        std::cout << numeros[i] << ", ";
    // No colocar coma al ultimo elemento
    std::cout << numeros[CANTIDAD_DE_ELEM - 1]
                << " es "
                << suma / CANTIDAD_DE_ELEM << '\n';
}
```

Podemos recorrer los elementos de un arreglo utilizando lo que se conoce como “**for basado en rango**” (**iteradores**). Supongamos que **vec** es un vector de enteros:

```
for (int n : vec)  
    std::cout << n << ' ' ;
```


Un vector es un objeto distinto a objetos de los tipos primitivos, su tipo es definido por el usuario mediante el constructor class (que NO veremos), y tiene asociado un conjunto de funciones especiales que llamamos métodos, aplicables a él.

Si **f ()** es un método, que puede tener parámetros, asociado a un objeto **obj**, podemos ejecutar el método “sobre el objeto **obj**” de esta forma:

obj.f()

En realidad es aplicar la función **f()** donde uno de sus parámetros es **obj**

El tipo **vector** posee una serie de métodos que pueden ser aplicados a un objeto tipo **vector**, por ejemplo:

- **push_back**—inserta un nuevo elemento al final del vector
- **pop_back**—elimina el último elemento del vector
- **operator[]**—da acceso a un elemento en una determinada posición del vector
- **at**—da acceso a un elemento en una posición dada y verifica que la posición es válida
- **size**—devuelve el número de elementos del vector
- **empty**—devuelve true si y sólo si el vector no posee elementos
- **clear**—convierte un vector en un vector vacío
- **operator =** —operador de asignación. Asigna un vector a otro vector

Ejemplo de uso de los métodos y funciones con vectores:

```
int main() {
    std::vector<int> lista; // declara un vector vacío, sin elementos
    // Coloco en lista los números pares del 0 al 10
    for (int i = 0; i <= 10; i++)
        // colocar solo los numeros pares
        if (i % 2 == 0) lista.push_back(i);
    // Imprime el tamaño del vector lista
    std::cout << "Tamaño del vector lista= " << lista.size() << '\n';
    // Imprimir el contenido del vector
    imprimir(lista);
    // Calcule la suma de los elementos del vector e imprimirla
    std::cout << suma(lista) << '\n';
}
```

veamos funciones imprimir() y suma() en próxima diapositiva.....

Ejemplo de declaración y definición de funciones :

```
/*
 * imprimir(v)
 *   Imprime en contenido de un vector de
enteros
 *   v es el vector a imprimir
 */
void imprimir(std::vector<int> v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}
```

```
/*
 * suma(v)
 *   suma los elementos de un vector
 *   v es el vector a sumar
 *   devuelve la suma de los elementos de v
 */
int suma(std::vector<int> v) {
    int result = 0;
    for (int elem : v)
        result += elem;
    return result;
}
```

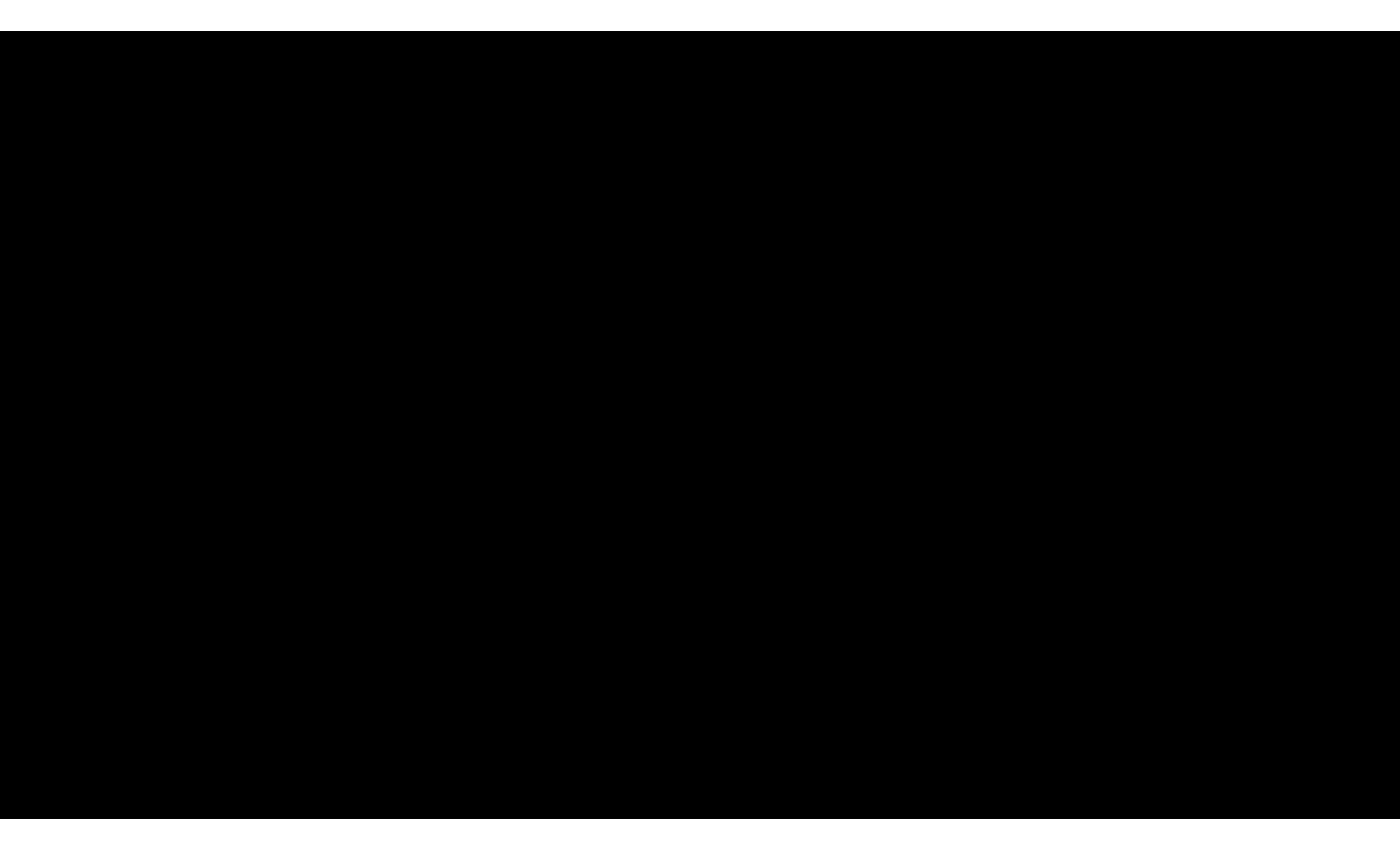
**Paso de
parámetros por
valor**



Hagamos juntos un programa que lea un vector de enteros, lo invierta e imprima el vector invertido.

Si el vector tiene: 1 2 3 4

Lo convierte en: 4 3 2 1



Solución:

```
#include <iostream>
#include <vector>
#include <limits>

/*
 *  imprimir(v)
 *      Imprime en contenido de un vector de enteros
 *      v es el vector a imprimir
 */
void imprimir_vector(std::vector<int> v) {
    for (int elem : v)
        std::cout << elem << " ";
    std::cout << '\n';
}
```

```

/*
 * leer_vector( )
 *     Lee un vector de enteros
 *     no tiene parámetros
 *     Devuelve un vector de enteros leído
 */
std::vector<int> leer_vector(){    // devuelve un vector!!!
    int valor;
    std::vector<int> vector;
    std::cout << "Coloque los elementos "
        << "del vector y termine con la palabra fin: \n";
    while (std::cin>>valor) vector.push_back(valor);
    // Limpia el estado de error de la entrada
    // de datos por teclado
    std::cin.clear();
    // Vacía el buffer de entrada
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    return vector;
}

```



```
int main() {

    std::vector<int> vector = leer_vector();
    int n = vector_size();
    for (int i = 0 ; i <= (n-1)/2; i++)
    {
        int temporal;
        // intercambiar valores
        temporal = vector[i];
        vector[i]= vector[n-1-i];
        vector[n-1-i] = temporal;
    }
    imprimir_vector(vector);
    return 0;
}
```

Una función puede devolver un vector:

```
/*
 * primos(comienzo, fin)
 *   Devuelve un vector conteniendo
 *   los números primos en el rango comienzo..fin.
 *   comienzo es el primer número en el rango
 *   fin es el ultimo número en el rango
 */
std::vector<int> primos(int comienzo, int fin) {
    std::vector<int> result;
    for (int i = comienzo; i <= fin; i++)
        if (es_primo(i))
            result.push_back(i);
    return result;
}
```

Como poder leer y escribir caracteres del español por línea de comandos en windows:

```
#include <iostream>
#include <windows.h>
```

```
int main() {
    SetConsoleCP(1252); // Cambiar STDIN - Para máquinas Windows
                        // y poder leer e imprimir ñ á, etc
    SetConsoleOutputCP(1252); // Cambiar STDOUT - Para máquinas Windows

    char xx;
    std::cin >> xx;
    std::cout << xx << std::endl;
    std::cout << "Tamaññño del vector = " << '\n';
}
```

Vectores multidimensionales: vector de vectores.

Una matriz 2 X 3:

```
std::vector<std::vector<int>> a(2, std::vector<int>(3));
```

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

```
a[0][1] = 19;
```

```
a[0][2] = 3;
```

```
a[1][0] = 22;
```

```
a[1][1] = -8;
```

```
a[1][2] = 10;
```

```
std::vector<std::vector<int>> a{{ 5, 19, 3},  
                                {22, -8, 10}};
```


Podemos simplificar el nombre del tipo colocando al comienzo del programa:

```
using Matriz = std::vector<std::vector<double>>;
```

Y, por ejemplo, podemos crear una función para imprimir una matriz:

```
void imprimir (const Matriz m){  
    for (int i = 0; i < m.size(); i++)  
        for (int j = 0; j < m[i].size(); j++)  
            std::cout << std::setw(5) << m[i][j];  
    std::cout << "\n";  
}
```

`#include <iomanip>`



○ también usando el for basado en rango:

```
void imprimir1 (const Matriz m){  
    for (auto fila: m)  
        for (auto elem: fila)  
            std::cout << std::setw(5) << elem;  
    std::cout << "\n";  
}
```

Hagamos juntos:

Leer fila a fila una matriz $n \times n$ de enteros e imprimir su diagonal. Suponga que lee primero n , la dimensión de la matriz



Solución:

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<vector<double>> Matriz;
// tambien: using Matriz = std::vector<std::vector<double>>;
Matriz leer_matriz( ){
    int n;
    std::cout << "Coloque la dimensión de la matriz: ";
    std::cin >> n;
    Matriz X (n,vector<double>(n));
    for (int i=0; i < n; i++) {
        cout<<"Coloque los elementos de la fila "
            << i <<" separados por espacio:"<<endl;
        for (int j=0; j<n; j++ )
            cin >> X[i][j] ;
    }
    return X;
}
```

```
void imprimir_diagonal(Matriz X){
    for (int i=0; i<X.size()-1; i++ )
        cout << X[i][i] <<" , ";
    cout << X[X.size()-1][X.size()-1] <<endl;
}
int main(){
    Matriz X = leer_matriz();
    imprimir_diagonal(X);
}
```

El tipo **string** forma parte de la biblioteca estándar de C++.

Permite almacenar una cadena de caracteres.

Un **string** es un objeto de C++ cuya estructura de datos que lo representa es un arreglo de caracteres (la forma de representar strings en C).

<https://www.geeksforgeeks.org/strings-in-cpp/>

```
#include <string>
```

```
....
```

```
using std::string; // para evitar colocar std:: de prefijo
```

```
string nombre1 = "jose", nombre2;
```

```
std::cout << nombre1 << '\n';
```

```
nombre2 = "oscar meza"; // tener cuidado con “ , debe ser "
```

```
std::cout << “la letra en el índice 0 es “ << nombre2 [0] << '\n';
```

Algunos operadores (o métodos) del tipo **string**:

- **operador[]** —permite acceso a un caracter en un indice dado
- **operador=** —asignar un string a otro
- **operador+=** —concatenar un string a otro
- **length** —devuelve el número de caracteres en el string
- **size** —igual que length
- **find** —localiza el índice de un sub-string en un string
- **substr** —devuelve un string que sea substring de un string
- **starts_with** —devuelve verdad sii un string es prefijo de un string
- **ends_with** —devuelve verdad sii un string es sufijo de un string
- **empty** —devuelve verdad si el string es vacío
- **clear** —convierte en vacío un string

Ejemplos de como operar sobre strings:

```
int main() {
    std::string palabra; // declara un string vacío
    std::cin >> palabra; // lee una cadena de caracteres sin espacios
    std::cout << palabra << " largo = " << palabra.length() << '\n';
    std::cout << palabra[palabra.length() - 1] << '\n';
    palabra += "fin"; // concatena lo que tenia palabra con la cadena "fin"
    if (palabra.empty()) std::cout << "vacío\n";
    else std::cout << "no vacío\n";
    palabra.clear();
    std::string aa[3] {"a", "bb", "c"}; // arreglo de strings
}
```

Otros ejemplos de métodos del tipo string:

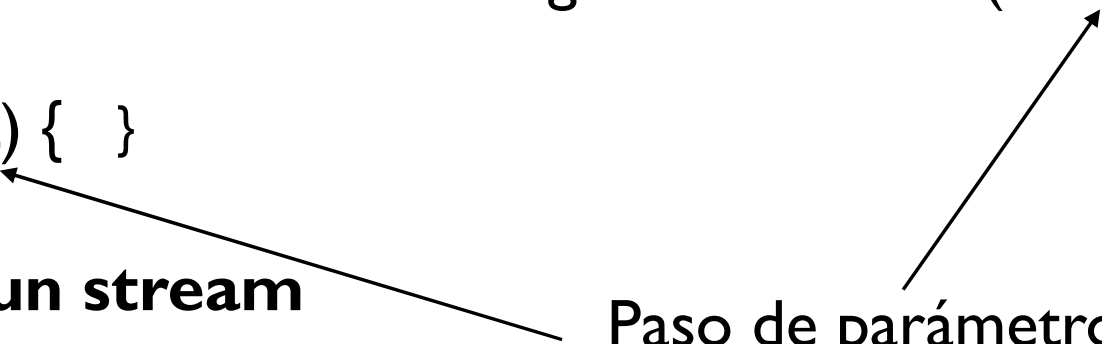
```
string string1 = "Beginner ";
string string2 = "to Expert ";
string string3 = "Tutorials";
string string4 = string1 + string2 + string3;
cout << "Expert en posición " << string2.find("Expert") << endl;
cout << "Parte del string 2: " << string2.substr(3,8) << endl;
cout << "Reemplazar 'Expert': "
           << string4.replace(12, 16, "Guru") << endl;
cout << "Inserción: " << string4.insert(0, " by Kindson") << endl;
cout << "Borrado: " << string3.erase(0,3) << endl;
```

Dividir en palabras, una cadena de caracteres leída con espacios:

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
using std::vector;
using std::string;
vector<string> Extraer(const string Text) { }
    vector<string> Words;
    std::stringstream ss(Text); // ss es un stream
    string Buf;
    while (ss >> Buf) // mientras existan palabras
        Words.push_back(Buf);
    return Words;
}
```

```
int main() {
    string linea ;
    std::cout << "Coloque una linea de texto: ";
    getline(std::cin, linea);
    vector<string> xx = Extraer(linea);
}
```

Paso de parámetro por
VALOR



PREGUNTAS???

