

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

Continuamos con vectores, funciones....

Una función puede devolver un vector

```

/*
 * primos(comienzo, fin)
 *   Devuelve un vector conteniendo
 *   los números primos que están 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;
}

```

Sobre el ejercicio de la última clase de invertir un vector:

Si hacemos una función que invierta un vector CUIDADO!! porque solo hasta el momento lo podemos pasar por valor y si lo invertimos en la función, no lo cambiará en el programa principal

Habría que devolverlo en la función y asignarlo al vector original en el programa que llamó a la función

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}};
```

```
int n =20, m=30;
```

Una matriz n X m:

```
std::vector<std::vector<double>> vec (n, std::vector<double>> (m));
```

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";
}
```

Indica que la variable **m** no
se puede modificar

#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:

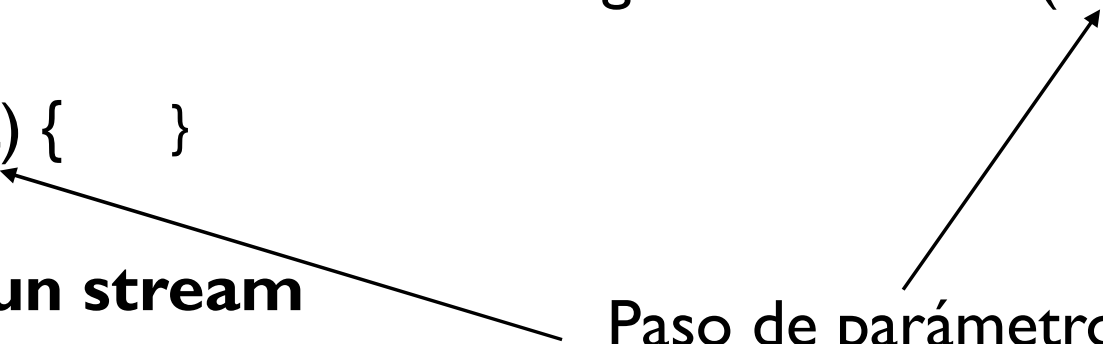
```
string string1 = "Beginner ";
string string2 = "to Expert ";
string string3 = "Tutorials";
string string4 = string1 + string2 + string3;
cout << "Expert en posición " << string2.find("Expert") << endl;
cout << "Parte del string 2: " << string2.substr(3,8) << endl;
cout << "Reemplazar 'Expert': "
           << 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



TIPOS DE DATOS ABSTRACTOS (ABSTRACT DATA TYPES)

- Un **Tipo de Dato Abstracto** es una abstracción matemática dada por un conjunto de **valores** (objetos) junto a un **conjunto de operaciones** que se pueden realizar sobre estos. Término que abreviaremos como **TDA**.
- **Por ejemplo, los números enteros** con sus operaciones de suma, resta, división, etc., es un tipo de dato abstracto que en los lenguajes de programación ya tienen una representación concreta. **Las secuencias, puntos en el plano cartesiano**, son tipos de datos abstractos.

- **Podemos valernos del “constructo” struct en C++ y funciones asociadas al tipo definido por el struct para implementar un Tipo de Dato Abstracto**
- **La construcción de un nuevo tipo de datos requiere que las características deseadas de éste sean inicialmente especificadas de manera clara, para luego proceder a implementarlo y obtener un tipo concreto de dato.**
- **El nombre TDAs se debe a que el comportamiento deseado de estos tipos debe ser especificado de manera abstracta, independientemente de las múltiples posibles implementaciones concretas que puedan luego construirse.**

- Para especificar (o representar) **TIPOS DE DATOS ABSTRACTOS** utilizamos **modelos matemáticos conocidos**, como por ejemplo, conjuntos, secuencias, funciones y relaciones.
- Los TDAs permiten, al momento de su implementación en C++, **reforzar los conceptos “encapsulamiento de datos”** (los datos junto a sus operaciones) **y en consecuencia el “ocultamiento de datos”** (sabemos qué hace y no cómo se hace, se ocultan detalles de implementación), dos principios básicos de la programación orientada a objetos que utilizaremos en este curso.
- Para ello nos valdremos de la separación en módulos de nuestros programas. Como veremos mas adelante crearemos archivos HEADER (.h) y su correspondiente .cpp

Ejemplo de tipos de datos abstractos:

- **Puntos** de un plano cartesiano y sus operaciones: distancia, primera coordenada, etc.
- **Secuencias** o lista de elementos y sus operaciones
- **Conjuntos** de elementos de un tipo dado y sus operaciones: unión, intersección, etc.

Una **struct** sirve agrupar un conjunto de datos:

Implementemos el tipo punto en el plano cartesiano:

```
// La struct Punto implementa el modelo matemático
```

```
// puntos del plano cartesiano
```

```
struct Punto {
```

```
    double x;  // primera coordenada
```

```
    double y;  // segunda coordenada
```

```
};
```

Algunas operaciones de; tipo Punto:

En <cmath>

```
double distancia(Punto p1, Punto p2){
    //double a1 = (p1.x - p2.x);
    //double a2 = (p1.y - p2.y);
    return (sqrt( pow((p1.x - p2.x), 2) +
                  pow((p1.y - p2.y), 2) ) );
}

double primera_coordenada(Punto p){
    return p.x; // se utiliza el "." para acceder a componentes de un struct
}

void imprimir(Punto p){
    std::cout << "(" << p.x << "," << p.y << ")" << std::endl;
}

31/3/2025}
```

Podemos utilizar el tipo Punto en nuestro programa:

```
int main() {

    // Declarar dos objetos tipo Punto
    Punto pt1, pt2; // crea dos objetos tipo Punto
    // Asignar sus coordenadas
    pt1.x = 8.5; // se utiliza el "." para acceder a componentes de un struct
    pt1.y = 0.0;
    pt2.x = -4;
    pt2.y = 2.5;
    Punto pt3 {1.5, 2.3}; //podemos declarar e inicializar un punto
    // Imprimirlos
    imprimir(pt1);
    imprimir(pt2);
```

¿Qué imprime?.....

La asignación:

pt1 = pt2

Permite **copiar** un objeto **struct** a otro sin necesidad de copiar por separado cada uno de sus datos. Todos los datos de un objeto serán asignados al otro objeto

Podemos definir un vector de Puntos:

```
std::vector<Punto> puntos(5000);
```

Podemos inicializar vectores de puntos:

```
struct Punto {
    double x ;
    double y;
};
```

```
std::vector<Punto> puntos {{2.4, 4.5},
{4.5, 7.4}};
```

```
std::cout << puntos[0].x << " " <<
puntos[1].y << '\n';
```

Imprime:

2.4 7.4

En resumen: con el constructo **struct** podemos implementar nuestros propios tipos de datos

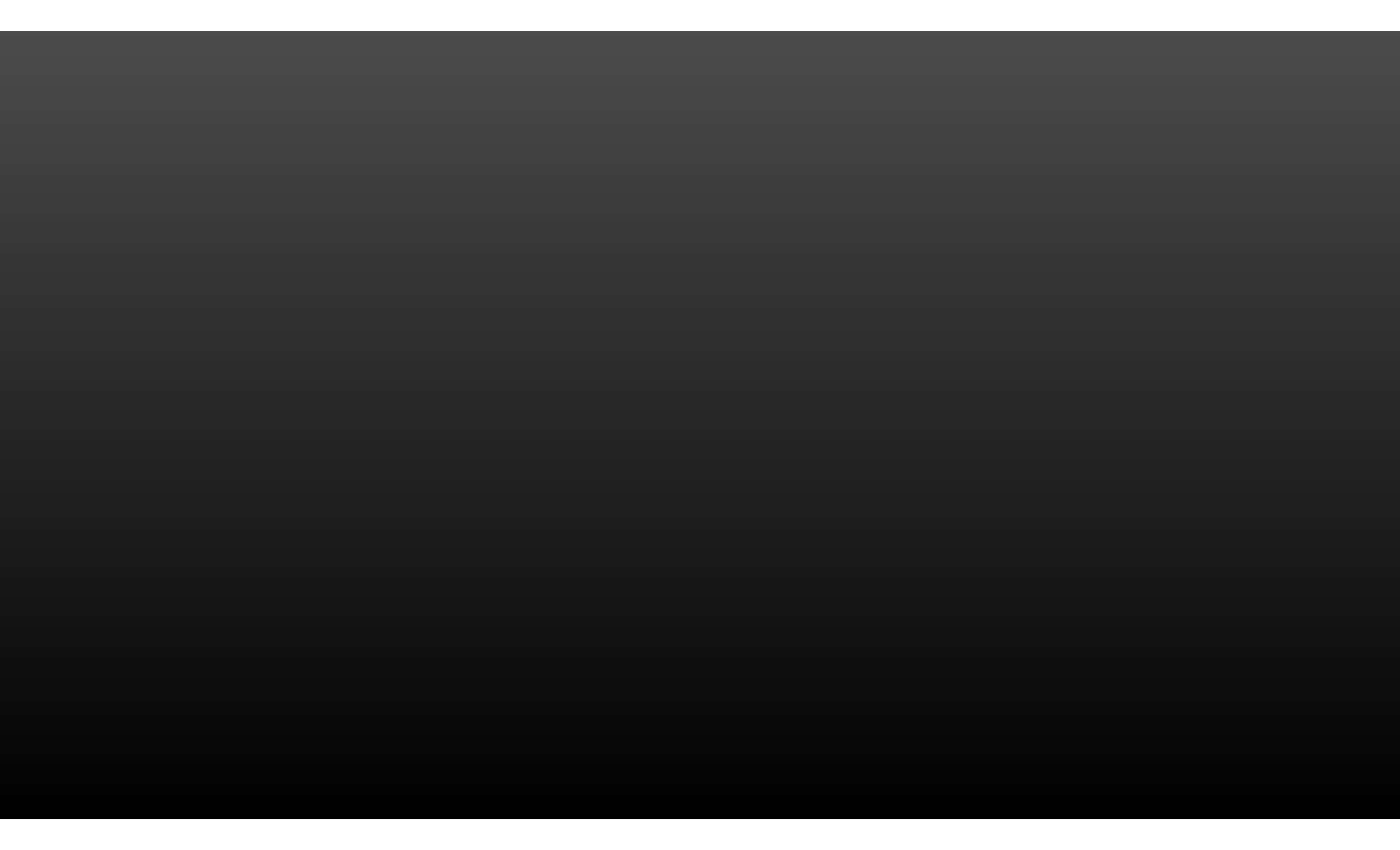
Esto lo haremos y aprenderemos a lo largo del curso

Ejemplo de uso de vectores:

Explicar: Ordenamiento de un vector de enteros por el método de selección

Hacer un ejemplo juntos en la pizarra

y luego veremos un programa



Ejemplo de un programa que ordena los puntos en un vector en forma creciente por la primera coordenada. Utiliza el algoritmo de ordenamiento por selección:

```
#include <iostream>
#include <vector>
// Ordenar por selección los puntos ascendentemente
// por primera coordenada un vector de puntos

int main ( ) {
    std::vector<Punto> puntos {{2,1}, {1,2}, {0,3}};
    int size = puntos.size();
    for (int i = 0; i < size - 1; i++) {
        int menor = i;
        for (int j = i + 1; j < size; j++)
            if (puntos[j].x < puntos[menor].x)
                menor = j;
        if (menor != i) {
            // intercambiar puntos
            Punto temp = puntos[i];
            puntos[i] = puntos[menor];
            puntos[menor] = temp;
        }
    }
}
```

Podemos hacer a las **funciones reusables**. Para poder utilizarlas en cualquier programa sin incluir su código, Y así modularizar nuestros programas para que sean más estructurados y fáciles de entender.

Veamos....

Creamos dos archivos, por ejemplo con el nombre **Archivo** (puede ser otro nombre): **Archivo.h** y otro **Archivo.cpp**

Archivo.h contendrá solo las estructuras de datos y el prototipo o declaraciones de las funciones (la primera línea de una función)

Archivo.cpp contendrá las definiciones de las funciones (el código en sí)

De esta forma podemos compilar nuestros programas que utilicen estas funciones, solo colocando la directiva `#include "Archivo.h"` al comienzo del programa

Ejemplo: creemos una función que determina si un número es primo y luego la utilizaremos en otros programas.

Creamos el archivo **primo.h** que contendrá solamente (no hay estructuras de datos):

bool es_primo(int); // prototipo de la función

Y luego creamos el **archivo primo.cpp** que contendrá:

Siguiente lámina...

```
// primo.cpp
#include <cmath>    // necesaria por sqrt( )
#include "primo.h" // el prototipo de _primo
/*
 * es_primo(n)
 *   Determina si un numero es primo
 *   n es el entero a verificar que es primo
 *   n deber mayor que 1
 *   Devuelve verdad si y solo si n es primo
 */
bool es_primo(int n) {
    bool result = true; // De antemano lo suponemos primo
    double r = n, raiz = sqrt(r);
    // Probamos con todos los posibles factores desde 2 hasta root
    for (int posible_factor = 2; result && posible_factor <= raiz;
        posible_factor++)
        result = (n % posible_factor != 0);
    return result;
}
```

Y utilizamos la función “es_primo” es nuestro programa “prueba_primo.cpp”:

```
#include <iostream>
#include "primo.h"
/*
 * main
 * Prueba la primalidad de los enteros comenzando desde 2
 * hasta un valor dado por pantalla.
 * Si el entero es un primo, lo imprime
 */
int main() {
    int max_value;
    std::cout << "Mostrar los primos hasta que valor?: ";
    std::cin >> max_value;
    for (int value = 2; value <= max_value; value++)
        if (es_primo(value)) // See if value is prime
            std::cout << value << " "; // Imprime el numero primo
    std::cout << '\n'; // mueve el cursor a la linea siguiente
}
```

Podemos compilar nuestro programa por línea de comando (en un terminal de VScode):

```
g++ -o prueba_primo prueba_primo.cpp primo.cpp
```

Y ejecutarlo:

prueba_primo (en ventana de comandos de Windows)
o
.\prueba_primo (en terminal VS Code)

Veremos en laboratorio como configuramos Visual Studio Code para no compilar por la terminal cuando hay archivos .h y otros .cpp además del .cpp que contiene el main() :

Será cambiar el task.json : "\${file}", por "\${workspaceFolder}*.cpp",

Otro ejemplo: Volvamos reutilizable el tipo Punto, incorporamos tres funciones: distancia, primera_coordenada e imprimir un punto

Creamos un archivo **punto.h** con el siguiente contenido:

```
// Punto implementa el modelo matemático
// puntos del plano cartesiano

struct Punto {
    double x;
    double y;
};

double distancia(Punto , Punto ); // el prototipo
void imprimir(Punto );
double primera_coordenada(Punto );
```

Luego creamos el archivo `punto.cpp` asociado a Punto, donde definimos las funciones:

```
#include <iostream>
#include <cmath>
#include "punto.h"
double distancia(Punto p1, Punto p2){
    return (sqrt( pow(p1.x – p2.x, 2) +
                  pow(p1.y – p2.y, 2) ) )
}
void imprimir(Punto p){
    std::cout << "(" << p.x << ", " << p.y << ")";
}
double primera_coordenada(Punto p){
    return p.x;
}
```


**Hagamos un programa que lea dos puntos
e imprima la distancia entre ellos
utilizando “punto.h”:**



Hagamos un programa que lea dos puntos e imprima distancia entre ellos:

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

int main(){
    Point p1, p2;
    cin >> p1.x >> p1.y>>p2.x>>p2.y;
    std::cout << " la distancia es: " << distancia(p1,p2) << std::endl;
}
```

Importante:

La única forma de **crear una función para el tipo Punto que inicialice las coordenadas de un punto** es que pasemos las coordenadas, se cree un punto dentro de la función al que le asignamos las coordenadas y luego devolvemos el punto creado (el problema del paso por valor).

Evitaremos esta función tan costosa cuando veamos **paso de parámetros por referencia**

Con este tipo de inicialización “**ocultaríamos**” como se define un Punto, es lo que se conoce como **ENCAPSULAMIENTO DE DATOS**

Veamos....

Agregar a punto.h :

```
Punto inicializar(double x, double y)
```

Agregar a punto.cpp :

```
Punto inicializar(double x, double y){  
    Punto p;  
    p.x = x;  
    p.y = y;  
    return p;  
}
```

Utilización en un programa:

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

int main(){ // note que no sabemos como se
            // representa (est. De datos) un punto
    Punto p1, p2;
    double x1, x2, y1, y2;
    std::cin >> x1 >> y1 >> x2 >> y2;

    p1 = Inicializar(x1, y1);
    p2 = Inicializar(x2, y2);
    imprimir(p1);
    imprimir(p2);
}
```

Hagamos la implementación del tipo conjunto de enteros **con header files.**

Utilice un vector para almacenar un conjunto de enteros y defina su tipo como:

```
using Conjunto_int = std::vector<int> ;  
ó  
typedef std::vector<int> Conjunto_int;
```

Las operaciones serán:

- **Insertar (agregar) un elemento.**
- **Verificar si un elemento está en un conjunto.**
- **Imprimir los elementos de un conjunto.**

Archivo Conjunto.h:

```
#include <vector>
using std::vector;
typedef vector<int> Conjunto_int;
Conjunto_int insertar(Conjunto_int , int );
bool pertenece(Conjunto_int , int );
void imprimir(Conjunto_int);
```


Archivo Conjunto.cpp: (no es necesario incluir `#include <vector>` pues está en el .h)

```
#include "Conjunto_int.h"
```

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

Y la definición de pertenece e imprimir....

Archivo prueba_conjunto.cpp (note que por encapsulamiento de datos no tenemos que conocer como se implementa Conjunto_int, solo utilizar sus operaciones):

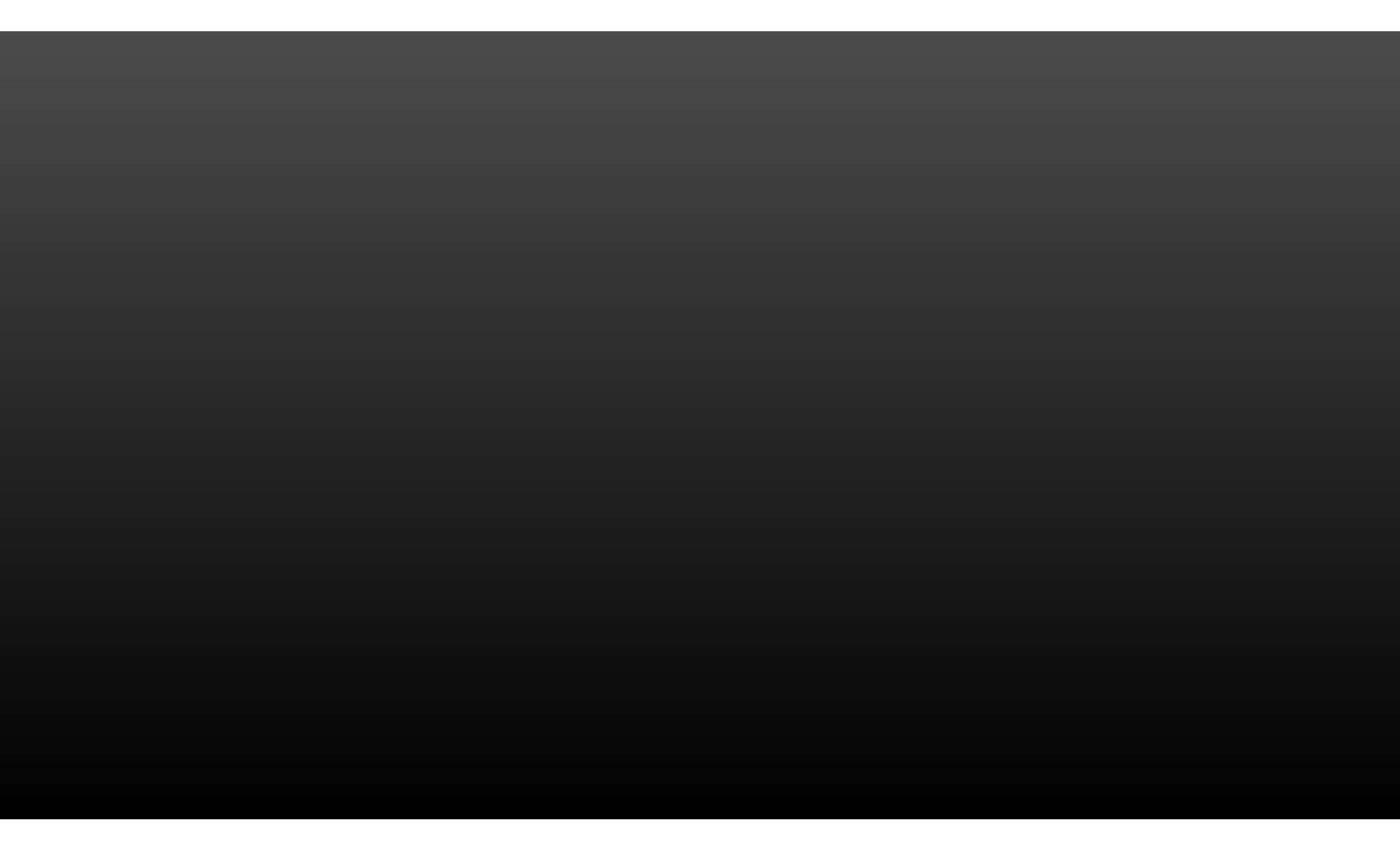
```
#include <iostream>
#include "Conjunto_int.h"
```

```
int main(){
    Conjunto_int conj;
    conj = insertar(conj,0); // note que devolvemos el conjunto modificado
    conj = insertar(conj,1);
    conj = insertar(conj,2);
    imprimir(conj);
}
```

Hagamos juntos pertenece e imprimir

bool pertenece(Conjunto_int , int);

void imprimir(Conjunto_int);



```
bool pertenece(Conjunto_int v, int i){  
    // devuelve verdad si y solo i esta en v  
    for (int j=0; j<v.size(); j++){  
        if (i==v[j]) return true;  
    }  
    return false;  
}
```

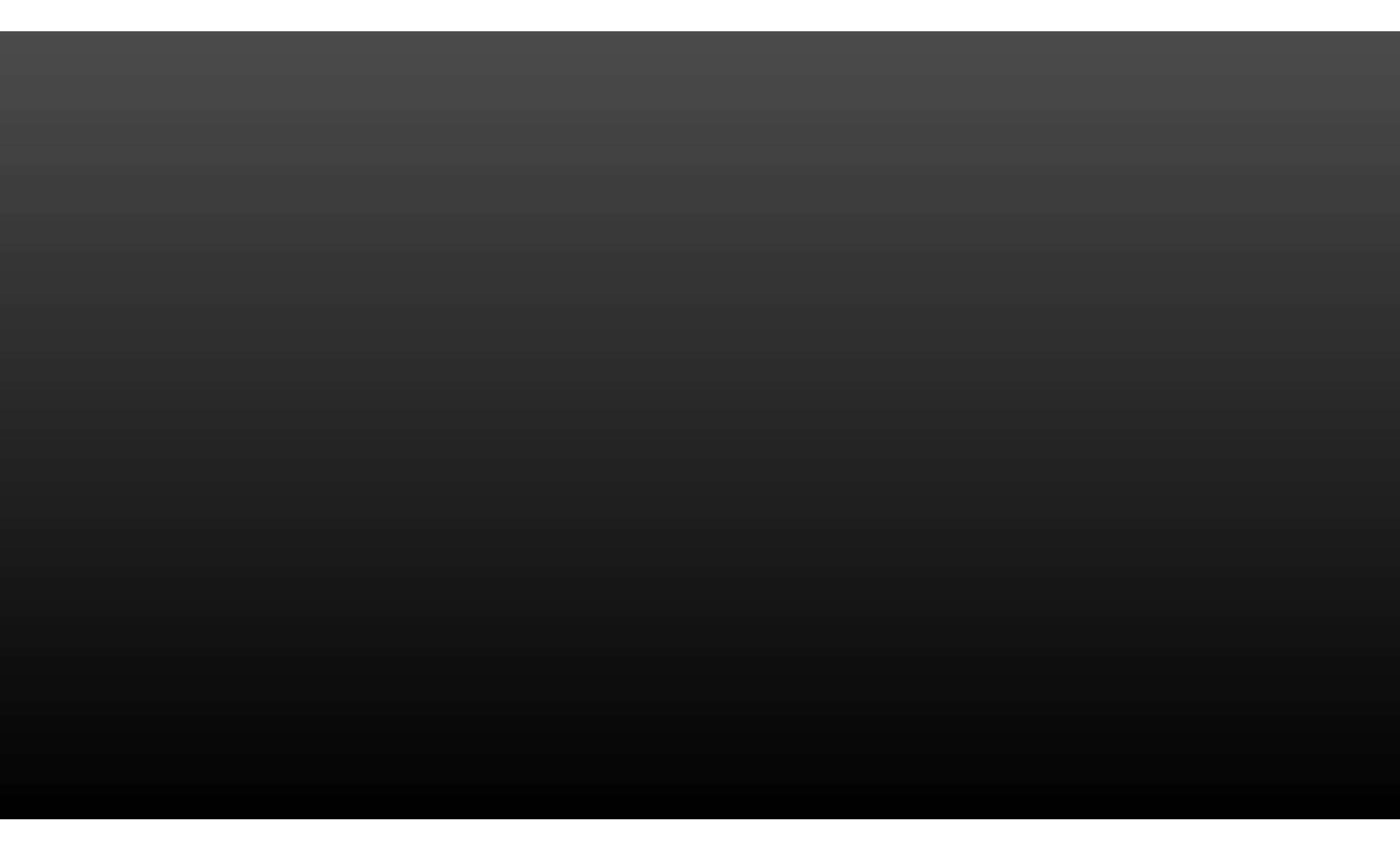
```
void imprimir(Conjunto_int conj){  
    for (int j=0; j<conj.size(); j++)  
        std::cout << conj[j] << " ";  
}
```

Podemos definir otras operaciones sobre el tipo Conjunto_int como son union, intersección, etc.

Hagamos juntos Union:

Conjunto_int unir(Conjunto_int, Conjunto_int)

Se deberán utilizar los métodos de tipo vector.. Ej: push_back



```
Conjunto_int unir(Conjunto_int A, Conjunto_int B){  
  
    Conjunto_int result;  
    for (auto i : A)  
        result.push_back(i);  
    for (auto i : B)  
        if (! pertenece(A,i))  
            result.push_back(i);  
    return result;  
  
}
```


Archivos de texto de lectura y escritura

Como crear un Archivo de texto de salida (crear un archivo de texto):

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    int entero;
    std::string filename;
    std::cout << " Nombre del archivo:";
    std::cin >> filename;
    // crear archivo
    std::ofstream out (filename); //crea out y lo
    // asocia al archivo (lo abre)

    if (out.is_open()) {    // si se abrio bien el archivo
        // colocar varios enteros y terminar con un caracter
        while (std::cin>>entero){
            // termina si coloco un carácter,
            // lo hace por un error al leer un caracter.
            //Habría que "resetear "cin" para utilizarlo después
            out << entero << " ";
        }
    } else    std::cout << "No se pudo escribir\n";
    out.close();
    return 0;
}
```

Leer un Archivo de texto de entrada (leemos un archivo):

```
std::cout << " Coloque el nombre del archivo: ";
std::cin >> filename;
std::ifstream in (filename);
int value;
if (in.is_open()) { // verificar archivo abierto
    while (in >> value) // leer hasta fin de archivo
        std::cout << value << "\n";
}
else std::cout << "No se puede abrir el archivo\n";
in.close();
```

Como acceder a un Archivo de texto de entrada (leemos un archivo):

Se puede utilizar:

```
string sa;  
getline(in, sa)
```

Y obtener una línea completa del archivo

PREGUNTAS???

