Algoritmos: Su propósito y el de las estructuras de datos es la resolución de problemas computables de información, por lo que se requiere la conjunción de:

→ Datos: definiendo sus tipos.
→ En una secuencia lógica y precisa de acciones (algoritmo).
→ Una solución implementada mediante lenguajes de programación.

Resolución de Problemas: encontramos diferentes etapas.

→ **Análisis del problema:** al leer el enunciado debemos entenderlo inmerso en un contexto.
→ **Diseño de la solución:** modularizar el problema.
→ **Especificación del Algoritmo:** elegimos un algoritmo que solucione individualmente cada uno de los pequeños problemas.
→ **Escritura del programa:** escribimos el código del Algoritmo.
→ **Verificación:** ejecutamos el nuevo programa y validamos que funcione.

Conceptos

→ **Paradigma de Programación:** modelo conceptual que define cómo se debe escribir un programa en el ordenador. Es decir, un conjunto de principios, métodos y conceptos que actúan como guía para escribir y abordar el proceso de resolución de problemas. Trabajamos en este caso con un paradigma:
  - Imperativo: dar órdenes para ejecutar una serie de sentencias.
  - Modular: descomponer un problema en bloques menores.
  - Estructurado: utilizar determinadas estructuras de resolución.

→ **Programación:** proceso de creación de un conjunto de instrucciones para una computadora o dispositivo que le permite realizar una tarea o resolver un problema específico.
→ **Programa:** conjunto de instrucciones que un dispositivo sigue para realizar una tarea o resolver un problema específico.
→ **Lenguaje de programación:** conjunto de reglas, símbolos, palabras y sintaxis que se usan para escribir programas de ordenador. Permite al programador comunicarse con una computadora y darle un set de instrucciones específico.
→ **Imprimir:** mostrar en pantalla.

Dato, Información y Conocimiento

→ **Dato:** representación de un objeto del mundo real. Es concreto. Permite modelizar aspectos de un problema que se quiere resolver mediante una computadora. Por ej: semáforo en rojo.
→ **Información:** interpretación de datos. Por ej: la luz roja supone alerta.
→ **Conocimiento:** saber que acciones ejecutar en función de la información disponible. Por ej: saber que no tengo que moverme y esperar al verde.

Conceptos (C++)

Identificadores: es un "nombre" que utiliza el programador para identificar los elementos del código. Estos son:

→ **Variables:** un valor que puede ser modificado durante el proceso del programa y que tiene asociado un espacio de memoria.

```
int contador = 0
tipoDeDato identificador = valor asignado
```

→ **Constantes:** asociado a un espacio de memoria que tiene un valor fijo que no cambia ni puede ser cambiado durante todo el proceso de la ejecución del programa.

```
const int notaMaxima = 10
constante tipoDeDato identificador = valor asignado
```

→ **Funciones:** asociado a un espacio de memoria que contiene líneas de código que se especializan en realizar una acción determinada y pueden ser reutilizadas en diferentes partes del programa. Son útiles porque se puede reutilizar código, modular, fácil mantenimiento, abstracción, fácil de probar y depurar.

Expresión: es un conjunto de operadores y operandos que puede reducirse a un valor. Por ej: la expresión a + b puede reducirse a su resultado c.

Sentencia: una orden que se le da al sistema, terminan con (;). Por ej: `c = a + b;`

→ **Simples:** una acción única.
→ **Estructuradas:** sirven para controlar el flujo (for, if, etc.)
→ **Compuestas:** una o más sentencias que se tratan como una unidad. {Sentencia, sentencia,...}

Declarar y Definir

→ **Declarar:** la primera vez que se introduce un elemento en el código. La especificación de su nombre y que tipo de variable o función es. Reserva un espacio de memoria a la variable sin asignarle un valor específico.

```
int edad;
tipoDeDato identificador;

int suma (int a, int b);
tipoDeDato identFuncion (datos que requiere);
```

→ **Definir:** implementación real del elemento, se establece el valor inicial de la variable o el comportamiento de la función.

```
int edad = 13;
tipoDeDato identificador = valorInicial;
```

En el main: `suma(x,y);`

Definición:

```
int suma (int a, int b) {
   return a + b;
}
```

En el main: `identFuncion (datos que usará);`

Definición:

```
tipoDeDato identFuncion (datos que requiere) {
   return valorInicial;
}
```

Bibliotecas: permiten utilizar más comandos.
Tabular: agregar espacios al inicio de líneas de código para ayudar a la organización.

C++

Tipos de Datos

→ **Caracteres:** `char` (1 solo caracter) , `string` (una cadena de caracteres), `wchar_t`
→ **Numeros Enteros (permite unsigned):** `short`, `int` (más común), `long`
→ **Numeros Reales (permite unsigned):** `float`, `double`
→ **Booleanos:** `bool`
→ **Vacío:** `void`

Tipos de Operadores

cin >>	Dispositivo std de entrada, permite al usuario ingresar un valor
cout <<	Dispositivo de salida, imprime en pantalla
>>	Operador de extracción de flujo
+	Suma
++	+1 (Incremento)
-	Resta
--	-1 (Decremento)
*	Multiplicación
/	División
%	Operador módulo o resto
()	Operador para agrupar expresiones a*(b-c)
==	Operador de igualdad
>	Mayor
>=	Mayor igual
<	Menor
<=	Menor igual
!=	Operador de desigualdad
=	Operador de asignación
+=	Asignación y suma, x+= 3 es igual a x = x + 3
-=	Resta y asignación
*=	Multiplicación y asignación

/=	División y asignación

Precedencia de Operadores

→ **Asociatividad:** se opera de izquierda a derecha. Ej: x = 10 + 5 x 8 + 30 = 10 + (5 x 8) + 30 = 80
→ **Orden de Evaluación:** se priorizan operaciones con (). Ej: x = (10 + 5) x 8 + 30 = 150

Estructuras de Control

IF: "*Si es verdadera la condición (expresión), entonces ocurre la primera sentencia, sino la otra. Si son varias sentencias, se utiliza {}".*

```
if (expresión)
   sentencia;
else otra sentencia;
```

Ejemplos:

```
if (lado1 == lado2 == lado3)
   cout << "equilatero" << endl;
else cout << "isosceles" << endl;
```

```
else {
    cout << "Ingrese otro número:" << endl;
    cin >> numero;
}
```

WHILE: lo utilizamos para repetir una secuencia la cantidad de veces necesarias, no sabemos esa cantidad. Primero se valida la condición y luego se ejecuta.

"Mientras se cumpla la condición (expresión), se realiza la primera sentencia".

```
while (expresión)
     sentencia;
```

Ejemplos:

```
while (otra_accion == 1);
```

```
while (fread(&estudiante, sizeof(Estudiante), 1, archivo) == 1) {
      cout << "Estudiante encontrado:" << endl;
      cout << "Código: " << estudiante.codigo << endl;
}
```

DO-WHILE: al igual que while, repite una secuencia n cantidad de veces mientras se verifica la condición. Su particularidad es que primero ejecuta la acción y luego valida la condición, por lo que el bloque de código se ejecutará al menos una vez, independientemente de la veracidad de la condición.

"Realizar sentencia, mientras se cumpla la condición (expresión)".

```
do sentencia;
while (expresión);
```

Ejemplos:

```
do {
        ejecuto_accion ();
        cout << "Querés hacer otra cosa?" << endl;
        cout << "1. Si" << endl;
        cout << "2. No" << endl;
        cin >> otra_accion;
    } while (otra_accion == 1);
```

```
do
    cout << "El valor del contador es: " << contador << endl;
    while (contador <= 5);
```

FOR: repite una secuencia n veces hasta cumplirse una condición. Se debe saber conocer n.

"Repetir sentencia hasta que la condición, que empieza en valor inicial y se actualiza con cada repetición, deje de cumplirse".

```
for (valor inicial; condición; actualización) {
    sentencia;
}
```

"Repetir sentencia hasta que el contador i, que empieza en valor inicial y se actualiza con cada repetición, llegue a n/valor final".

```
for (int i = valor inicial; i < n; i++) {
    sentencia;
}
```

Ejemplos:

```
for(int i = 0; i < 5; i++) {
        Alumnos[i] = crearAlumno();
    };
```

SWITCH (SELECTOR): la utilizamos cuando se desea ejecutar diferentes bloques de código según la decisión del usuario.

"Según el valor de la expresión, en este caso la elección del usuario, se ejecuta el caso que tiene por etiqueta ese valor. El default es en caso que no se tenga una opción predeterminada para el valor elegido".

```
switch (expresión) {
    case valor1:
        sentencia;
        break;
    case valor2:
        sentencia;
        break;

    case valorn:
        sentencia;
        break;
    default:
        sentencia;
}
```

Ejemplos:

```
switch (eleccion_usuario ()){
   case 1:
   suma();
   break;

   case 2:
   triangulo();
   break;

   case 3:
   numeros_maximo();
   break;

   case 4:
   factorial();
   break;

   default:
   cout << "Opción Inválida" << endl;

   }
```

Funciones: bloques de código que realizan una tarea específica y pueden ser reutilizadas en diferentes partes de un programa.

Existen cuatro tipo de funciones:

- → Ingresa valores, retorna valores.
- → Ingresa valores, no retorna valores.
- → No ingresa valores, retorna valores.
- → No ingresa valores, no retorna valores.

Ventajas:

- → **Reutilización de Código:** como pueden ser reutilizadas en diferentes partes del programa, se reduce el código.
- → **Modularidad:** permiten dividir el programa en módulos más pequeños, volviéndolo fácil de entender y mantener.
- → **Fácil mantenimiento y depuración:** al modulizar, es más fácil identificar y corregir errores en secciones específicas del código. Se las aísla y depura de forma independiente.
- → **Abstracción:** oculta la complejidad del código y proporciona una interfaz sencilla y fácil de usar para el usuario.
- → **Facilidad de Prueba:** al ser unidades independientes, se pueden probar de forma individual. Facilita la tarea de prueba.

Declarar, Definir y Llamar a una Función

Declarar una Función: Antes de usar una función, hay que declararla. Esto provee información como el nombre de la función, el tipo de dato que devuelve (si corresponde) y los tipos y cantidad de parámetros que acepta.

```
tipoRetorno identFuncion(tipoDeDato ident1, tipoDeDato ident2, ...);
```

```
int suma(int a, int b);
```

Llamar a una Función: cuando utilizamos la función en el main, es decir ejecutamos el código de la función. Los argumentos que le pasamos deben coincidir con los tipos y cantidad declarados en la función.

```
tipoRetorno resultado = identFuncion(argumento1, argumento2, ...);
```

```
int resultado = suma(num1, num2);
```

Definir una Función: bloque de código que se ejecuta cuando la función es llamada.

```
tipoRetorno identFuncion(tipoDeDato ident1, tipoDeDato ident2, ...) {
    sentencia;
    return valorRetorno; // Opcional, si la función devuelve un valor
}
```

```
int suma(int a, int b) {
    return a + b;
}
```

Ejemplo: Suma Básica

```
#include <iostream>
using namespace std;

int suma(int a, int b);

int main() {
   int x = 5;
   int y = 7;

   int resultado = suma(x, y);
   cout << "La suma de " << x << " y " << y << " es " << resultado << endl;
   return 0;
}

int suma(int a, int b) {
return a + b;
}
```

Contexto de Ejecución de una Función: el ambiente con todos los elementos presentes (variables, parámetros, información necesaria para su ejecución) en el que se ejecuta una función. Cada función tiene su propio contexto de ejecución y cualquier cambio realizado dentro de ella no afectará al contexto fuera de ella.

El contexto de **main** es: x, y, resultado, suma.
El contexto de **suma** es: a, b.

_____

Parámetros de una Función

Paso por Valor: se pasa una copia del valor del parámetro a la función, lo que significa que cualquier cambio realizado dentro de la función no afectará al valor original del parámetro fuera de ella. Hay una diferencia entre aquello dentro de la memoria y aquello impreso en pantalla. Se utiliza cuando no se quiere modificar el valor original de una variable.

Por ej: si tuviera una función duplicadora e ingreso el número 5, se mostrará en pantalla el número 10, pero en memoria se mantendrá como 5. Si quisiera hacer otra acción sobre la variable, lo hará con 5.

```
void duplicar(int a)
```

Paso por Referencia: se pasa directamente la dirección de memoria del parámetro a la función. Esto significa que cualquier cambio realizado dentro de la función afecta el valor original del parámetro por fuera.

Por ej: teniendo la función duplicadora, el valor impreso en pantalla es 10 y el valor en memoria también es 10. Si se ejecuta otra acción sobre la variable, se lo hará al valor 10.

```
void duplicar(int& a)
```

Ejemplo:

```
#include <iostream>

void duplicar(int& a) {
  a = a * 2;
  cout << "Dentro de la función: " << a << endl;
}

int main() {
    int numero = 5;

   duplicar(numero);
   cout << "Fuera de la función: " << numero << std::endl;
   return 0;
}
```

Funciones Recursivas: son las funciones que se llaman a sí mismas dentro de su bloque de código. Permite dividir problemas complejos en una serie de pasos pequeños y manejables. Consta de un caso base, la condición que define el punto en el que la recursión se detiene, y el caso recursivo, en donde la función se llama a sí misma reduciendo gradualmente el problema al caso base. Necesitan si o si de una condición que permita detenerlas de convertirse en un loop infinito. Por ej: la función factorial, que se llama a sí misma con el argumento (n-1) hasta que se alcanza el caso base (n == 1), allí la recursión se detiene y el resultado final se devuelve.

```
int factorial(int n) {
  if (n == 1) {
   return 1;
  } else {
   return n * factorial(n - 1);
  }
}
```

Sobrecarga de Funciones: es una técnica que permite nombrar de igual forma a funciones con funcionalidad similar, pero que cuentan con diferentes parámetros de entrada (cantidad, tipo u orden de argumentos que reciben).

Cuando se llama a una función sobrecargada, el compilador determina automáticamente cuál de las funciones debe invocar basándose en los argumentos proporcionados. Simplifica el código y lo hace más fácil de entender. Por ejemplo: dos funciones sumas, uno con parámetros enteros y otros de punto flotante.

```
#include <iostream>

// Declaro funciones
int suma(int a, int b) {
  return a + b;
}

float suma(float a, float b) {
  return a + b;
}

int main() {
  int num1 = 5;
  int num2 = 10;
  cout << "La suma de " << num1 << " y " << num2 << " es " << suma(num1,
num2) << endl;

  float num3 = 3.14;
  float num4 = 2.71;
  cout << "La suma de " << num3 << " y " << num4 << " es " << suma(num3,
num4) << endl;
return 0;
}
```

Datos Derivados y Abstractos: Existen datos que no son primitivos, sino que derivan de otros tipos de datos. Estos datos se llaman datos derivados.

Struct: La estructura Struct permite crear nuevos tipos de datos para representar nuevas abstracciones, agrupando varios campos en una sola entidad. Le permite al programador crear sus propios tipos de datos, muchas veces conformados por dos o más datos primitivos. Por convención, se escribe con primera letra mayúscula.

Definir un Dato Derivado: se empieza definiendo la estructura struct, luego el nombre que se le asignará al nuevo tipo de dato. Entre llaves se definen los tipos de datos que conformarán al struct y por último se les asigna nombres.

```
struct NombreEstructura {
    TipoCampo1 nombreCampo1;
    TipoCampo2 nombreCampo2;
};
```

Por ejemplo:

```
struct Persona {
    int dni;
    string nombre;
};
```

Crear una Variable: `NombreEstructura identificador;`

```
Persona una_persona;
```

Asignar Valores a un Dato Derivado: se hace a través del operador punto y el nombre del campo a definir.

```
NombreEstructura identificador;

variable.nombreCampo1 = valorCampo1;
variable.nombreCampo2 = valorCampo2;
```

```
Persona una_persona;

una_persona.dni = 46501987;
una_persona.nombre = "Juan";
```

Completo:

```
struct NombreEstructura {
    TipoCampo1 nombreCampo1;
    TipoCampo2 nombreCampo2;
    // ...
};

NombreEstructura variable;

variable.nombreCampo1 = valorCampo1;
variable.nombreCampo2 = valorCampo2;
```

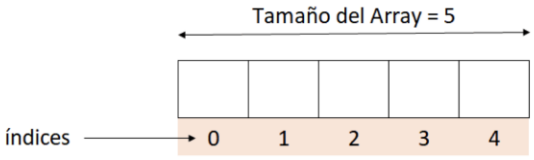
```
struct Persona {
    int dni;
    string nombre;
};

Persona una_persona;

una_persona.dni = 46501987;
una_persona.nombre = "Juan";
```

Estructuras Indexadas

Array: estructura de datos indexada que permite almacenar múltiples elementos del mismo tipo en una sola variable. Cada elemento se identifica por su posición en el Array, que empiezan desde el 0. Siempre va a haber un carácter nulo al final del Array.



Declaración:

```
tipoDeDato identArray[tamaño];
int numeros[5];
```

Definición de una posición: `numeros[0] = 10;`

Inicialización al declarar: En general definimos los valores de un Array al declararla. `int numeros[5] = {10, 20, 30, 40, 50};`. Si queremos hacerlo después es necesario utilizar:

```
#include <cstring> // Necesario para usar la función strcpy

int main() {
    char nuevoTexto[10];
    strcpy(nuevoTexto, "nuevo"); // strcpy cambia el valor existente del
                                    array
return 0;
}
```

Tamaño de un Array (Cantidad de Posiciones): podemos conocerlo a través de la función sizeof().

```
int tamaño = sizeof(array) / sizeof(array[0]);
```

Aquí sizeof(numeros) devuelve el tamaño en bytes del array, al dividirlo por sizeof(numeros[0]) que devuelve el tamaño en bytes de una sola posición, calcula su tamaño en términos de su cantidad de posiciones.

`int numeros[5]` → sizeof(`numeros`) = `5` * sizeof(int) → `5*sizeof(numeros[0])`

Ejemplo: se solicita al usuario ingresar 5 números que almacena en un array, e imprime en pantalla la suma de todos ellos.

```
#include <iostream>
using namespace std;

int main() {
  int numeros[5];
  int suma = 0;

  for(int i = 0; i < 5; i++) {
```

```
    cout << "Ingrese el número " << i + 1 << ": ";
    cin >> numeros[i];
  }

  for(int i = 0; i < 5; i++) {
    suma += numeros[i]; // suma = suma + el número de la posición i
  }

  cout << "La suma de los números ingresados es: " << suma << endl;

return 0;

}
```

Ejemplo en Clase: se solicita al usuario ingresar 5 alumnos que almacena en un array, e imprime luego en pantalla.

```
#include <iostream>
using namespace std;

// Declaro struct y función

struct Alumno {
  string nombre;
  int edad;
};

Alumno crearAlumno();

// Código Principal

int main() {
  Alumno alumnos[5];

  for(int i = 0; i < 5; i++) {
   alumnos[i] = crearAlumno();
    }

  for(int i = 0; i < 5; i++) {
   cout << "El alumno en la posición " << i + 1 << " tiene el nombre " <<
alumnos[i].nombre << " y su edad es " << alumnos[i].edad << endl;
    }

  return 0;
}

// Definición de Función

Alumno crearAlumno() {
```

```
    Alumno alumno;
    cout << "Ingrese el nombre del alumno: "<< endl;
    cin >> alumno.nombre;

    cout << "Ingrese la edad del alumno: "<< endl;
    cin >> alumno.edad;

    return alumno;
}
```

Arrays como Strings: Podemos pensar una cadena de caracteres como un Array de caracteres, donde su tamaño es el máximo de caracteres que queremos almacenar. Podemos elegir inicializarla al declarar.

```
char cadena[tamaño];
```

```
char cadena[tamaño] = "texto";
```

Si la cadena que estamos inicializando es más larga que el tamaño definido, el compilador automáticamente añade un carácter nulo (\0) al final de la cadena. Es útil para saber la cantidad de caracteres que tiene la frase original. Por ej: podemos contar mientras recorremos un array hasta llegar al carácter nulo para saber la cantidad.

```
#include <iostream>
using namespace std;

int main() {
  char cadena[100];
  cout << "Ingrese una cadena de caracteres: ";
  cin >> cadena;

  int tamaño = 0;
  while (cadena[tamaño] != '\0')
     tamaño++;

cout << "La cadena ingresada tiene " << tamaño << " caracteres." << endl;
return 0;
}
```

En este ejemplo, se utiliza un ciclo **while** para recorrer el array de caracteres **cadena**, y se incrementa la variable **tamaño** en cada iteración hasta encontrar el **carácter nulo**. Luego, se muestra por pantalla el tamaño final de la cadena obtenida.

Podemos acceder a un carácter puntual utilizando su índice: `char segundaLetra = cadena[1];`, o utilizar un bucle para recorrer todos los caracteres del Array:

```
for(int i = 0; i < tamaño; i++) {
    cout << cadena[i];
}
```

Matriz: estructura de datos que permite almacenar múltiples elementos en una tabla bidimensional. Cada elemento se identifica por su posición en la matriz, que empiezan desde el 0.

Declaración:
```
tipoDeDato identMatriz[numeroFilas][numeroColumnas];
int matriz[2][3];
```

Definición de una posición: `matriz[0][0] = 10;`

Inicialización al declarar: No se puede definir un Array luego de declararla, debe ser si o si en ese momento. `int matriz[2][3] = {{10, 20},{30, 40, 50}};`

Tamaño de una Matriz (Cantidad de Posiciones): podemos conocerlo a través de la función sizeof().
```
int numFilas = sizeof(matriz) / sizeof(matriz[0]);
int numColumnas = sizeof(matriz[0]) / sizeof(matriz[0][0]);
```

Aquí sizeof(matriz) devuelve el tamaño en bytes de la matriz, al dividirlo por sizeof(matriz[0]) que devuelve el tamaño en bytes de la primera fila de la matriz, calcula la cantidad de filas de la matriz. Para calcular la cantidad de columnas se divide sizeof(matriz[0]) por sizeof(matriz[0][0]), que devuelve el tamaño en bytes de un solo elemento de la matriz.

Ejemplo: se solicita al usuario ingresar 9 números que almacena en una matriz 3x3, e imprime en pantalla la suma de todos ellos.

```
#include <iostream>
using namespace std;

int main() {
  int matriz[3][3];
  int suma = 0;

  for(int i = 0; i < 3; i++) {// marca el comienzo de cada fila
    for(int j = 0; j < 3; j++) { // marca cada espacio, solo después de
                                    completarse los tres espacios de una
                                    fila, i++

    cout << "Ingrese el elemento [" << i << "][" << j << "]: ";

    cin >> matriz[i][j];
  }}

   for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++){
        suma += matriz[i][j];
  }}
     cout << "La suma de los elementos de la matriz es: " << suma << endl;
```

```
  return 0;
  }
```

Ejemplo 2: realizar la suma de dos matrices cuadradas de tamaño NxN.

```
#include <iostream>
using namespace std;

const int N = 3; // Tamaño de la matriz cuadrada

int main() {
  int matriz1[N][N];
  int matriz2[N][N];
  int resultado[N][N];

  cout << "Ingrese los valores de la primera matriz: " << endl;

  for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++){
     cin >> matriz1[i][j];
  }}

  cout << "Ingrese los valores de la segunda matriz: " << endl;
  for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++){
     cin >> matriz2[i][j];
  }}

  for (int i = 0; i < N; i++) {  // Sumar las matrices
   for (int j = 0; j < N; j++){
    resultado[i][j] = matriz1[i][j] + matriz2[i][j];
  }}

  cout << "La matriz resultante es: " << endl;
  for (int i = 0; i < N; i++) { // Mostrar el resultado
   for (int j = 0; j < N; j++){
    cout << resultado[i][j];
  }}
return 0;
}
```

Patrones Algorítmicos

Patrones de Carga:

1. **Carga Secuencial:** implica llenar un array secuencialmente, asignando valores a cada posición de forma consecutiva desde el inicio al final.

```
const int SIZE = 5;  // Tamaño del array
int array[SIZE];   // Declaración del array

for (int i = 0; i < SIZE; i++) {

   array[i] = i + 1;     // Asignar valor secuencial a cada posición
}
```

2. **Carga Directa:** asigna valores específicos a posiciones determinadas del Array sin seguir un orden.

```
int array[5];

array[0] = 10; // Asignar valor 10 a la primera posición
array[2] = 25;
array[4] = 50;
```

3. **Carga Ordenada:** asigna valores al Array en orden ascendente o descendente, utilizando un criterio específico.

```
const int SIZE = 5;
int array[SIZE];

for (int i = 0; i < SIZE; i++) {  // Orden Ascendente (0, 10, 20, 30, 40)
   array[i] = i * 10;
}

for (int i = 0; i < SIZE; i++) { // Orden Descendente (40, 30, 20, 10, 0)
   array[i] = (SIZE - 1 - i) * 10;
}
```

Patrones de Recorrido:

1. **Recorrido Total/Secuencial:** visita todos los elementos del Array en secuencia, desde el primer elemento hasta el último. Por ej: hacer un recorrido e imprimir todos los elementos.

```
const int SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5}; // Array inicializado

for (int i = 0; i < SIZE; i++) { // Realiza acción en cada elemento
   cout << array[i] << " ";
}

// Salida: 1 2 3 4 5
```

2. **Recorrido Parcial:** visita un subconjunto de elementos del Array, generalmente definido por un rango específico. Por ej: omitir primer y último elemento, e imprimir elementos restantes.

```
const int SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5};

for (int i = 1; i < SIZE - 1; i++) {
   cout << array[i] << " ";
}

// Salida: 2 3 4
```

3. **Recorrido en Ambas Direcciones:** visita los elementos del array tanto de forma ascendente como descendente, en diferentes etapas del recorrido. Por ej: imprimir elementos en ambos órdenes.

```
const int SIZE = 5;
int array[SIZE] = {1, 2, 3, 4, 5};


for (int i = 0; i < SIZE; i++) { // Recorrido ascendente, Salida: 1 2 3 4 5
   cout << array[i] << " ";
}

for (int i = SIZE - 1; i >= 0; i--){ //Recorrido descendente, Salida: 54321
   cout << array[i] << " ";
}
```

4. **Recorrido con Corte de Control:** implica detener el recorrido cuando se cumple una condición específica, en lugar de visitar todos los elementos del array. Por ej: el bucle recorre el array y se detiene al encontrar un valor igual al anterior.

```
const int SIZE = 5;

int array[SIZE] = {1, 2, 2, 3, 4};

for (int i = 1; i < SIZE; i++) { // Repetir acción hasta cumplir cond. de
                                    corte

  if (array[i] == array[i - 1]) { // Interrumpir recorrido cuando el valor
                                     sea igual al anterior

      break;

  }

cout << array[i] << " ";

  } // Realizar acción en elemento actual, por ej: imprimir su valor

// Salida: 2
```

Patrones de Búsqueda:

1. **Búsqueda Lineal/Secuencial:** busca un elemento en un array recorriendo secuencialmente cada elemento hasta encontrar una coincidencia.

```
bool busquedaLineal(const int array[], int size, int valorBuscado, int&
indiceEncontrado) {
  for (int i = 0; i < size; i++) {
       if (array[i] == valorBuscado) {
            indiceEncontrado = i;  // Almacena el índice donde se encontró
                                      el valor
            return true; // Valor encontrado
} }
   return false; // Valor no encontrado
}
```

Otra Forma:

```
int busquedaLineal(const int array[], int size, int valorBuscado) {
  for (int i = 0; i < size; i++) {
       if (array[i] == valorBuscado) {
            return i + 1; // Posición encontrada (1-indexed)

} }
   return -1; // Valor no encontrado
}
```

2. **Búsqueda Binaria:** se aplica cuando el **array está ordenado.** Se busca un valor específico dividiendo el rango de búsqueda a la mitad en cada iteración. No vale la pena ordenar un array para un uso único.

```
bool busquedaBinaria(const int array[], int size, int valorBuscado) {
  int inicio = 0;
  int fin = size - 1;

  while (inicio <= fin) {
       int medio = (inicio + fin) / 2;

       if (array[medio] == valorBuscado) {
            return true; // Valor encontrado

       } else if (array[medio] < valorBuscado) {
            inicio = medio + 1; // El valor está en la mitad derecha

       } else {
            fin = medio - 1; // El valor está en la mitad izquierda

} }
  return false; // Valor no encontrado
```

```
}
```

Otra Forma:

```
int busquedaBinaria(const int array[], int size, int valorBuscado) {
  int inicio = 0;
  int fin = size - 1;

  while (inicio <= fin) {
       int medio = (inicio + fin) / 2;

       if (array[medio] == valorBuscado) {
            return medio + 1; // Posición encontrada (1-indexed)

       } else if (array[medio] < valorBuscado) {
            inicio = medio + 1; // El valor está en la mitad derecha

       } else { fin = medio - 1; // El valor está en la mitad izquierda
} }
   return -1; // Valor no encontrado
}
```

Patrones de Ordenamiento:

1. **Simple:** ordena un array comparando pares de elementos adyacentes y realizando intercambios si están en el orden incorrecto. El proceso se repite hasta que el array está ordenado.

```
void ordenar(int array[], int SIZE) {
  int aux;
  int i, j, ord = 0;
  for(i = 0; i < SIZE - 1 && ord == 0; i++) {

   ord = 1;
    for(j = 0; j < SIZE - 1; j++) {

     if(array[j] > array[j + 1]) {
        aux = array[j];
        array[j] = array[j + 1];
        array[j + 1] = aux;
        ord = 0;
      }
    }
  }
}
```

Se establecen las variables de control `i` y `j`. `ord` se utiliza para verificar si se realizaron intercambios en una pasada completa del array.

El primer bucle `for` *itera desde* `i = 0` *hasta* `i < SIZE - 1`*, y verifica si* `ord == 0`*.* `ord` *se inicializa en 1 al comienzo de cada pasada, asumiendo que el array "ya fue ordenado". Si se llegará a realizar un intercambio dentro del segundo bucle,* `ord = 0`*, por lo que el bucle continua.*

El segundo bucle `for` *itera desde* `j = 0` *hasta* `j < SIZE - 1` *y compara cada elemento del array con su siguiente. Si el elemento es mayor al siguiente intercambian posiciones y* `ord = 0`*. Si se completa una pasada completa del array y no se realizan cambios (ord sigue siendo 1), significa que el array está ordenado y el bucle externo se detiene.*

2. **De Selección:** se comienza seleccionando repetidamente el elemento más pequeño y colocándolo en su posición correcta. A medida que avanzamos en el array, encontramos el elemento mínimo restante y lo intercambiamos con el elemento en la posición actual. El proceso continúa hasta que el array esté ordenado.

```
void ordenarPorSelección(int array[], int SIZE) {
  for (int i = 0; i < SIZE - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < SIZE; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }
        int temp = array[minIndex]; // Intercambio de elementos
        array[minIndex] = array[i];
        array[i] = temp;
        }
  }
```

El bucle `for` *externo recorre el array desde* `i = 0` *hasta* `i < SIZE - 1`*. Cada iteración encuentra el elemento más pequeño y lo coloca en su posición correcta. Declaramos dentro de él una variable* `minIndex` *que permite rastrear el índice del elemento mínimo encontrado hasta el momento, al comenzar suponemos que el elemento actual (*`array[i]`*) es el mínimo.*

El bucle `for` *interno recorre el array desde* `j = i + 1` *hasta* `j < SIZE`*, buscando un elemento más pequeño que el actual (*`array[i]`*) y actualiza* `minIndex` *con el índice del elemento más pequeño encontrado.*

Se realiza entonces un intercambio entre el elemento en la posición `i` *y el elemento más pequeño encontrado en la posición* `minIndex`*, a través de la variable* `temp` *(espacio temporal que permite intercambiar los valores sin perder el valor original de uno de los elementos).*

El bucle externo continúa su iteración para la siguiente posición, y el proceso se repite hasta que el Array esté ordenado.

3. **De Inserción:** se divide el array en una parte ordenada y una parte desordenada. En cada iteración, se toma un elemento de la parte desordenada y se inserta en la posición correcta dentro de la parte ordenada, desplazando los elementos mayores según sea necesario.

```
void insertionSort(int array[], int SIZE) {
  for (int i = 1; i < SIZE; i++) {
        int key = array[i];
        int j = i - 1;

        while (j >= 0 && array[j] > key) { // Mover los elementos mayores
                                              hacia la derecha
            array[j + 1] = array[j];
            j--;
        }

        array[j + 1] = key; // Insertar el elemento en su posición correcta
  }
}
```

El bucle `for` recorre el Array desde `i = 1` hasta `i < SIZE`. La idea es que el elemento en la posición `i` sea insertado en la parte ya ordenada del Array.

Se guarda el valor del elemento de la posición `i` en la variable `key`. Se declara una variable `j` que representa el índice del último elemento en la parte ordenada del Array (los elementos a la izquierda de la posición `i`).

El bucle `while` desplaza los elementos mayores hacia la derecha para hacer espacio al elemento `key` en la parte ordenada del Array. Se ejecuta mientras `j >= 0`, y el elemento de la posición `j > key`. En cada iteración se desplaza el elemento en la posición `j` a la derecha.

Ya hechos los desplazamientos necesarios, se inserta el elemento `key` en su posición correcta, que es `array[j + 1]`. La parte ordenada del Array se expande para incluir el nuevo elemento. Continua el bucle hasta que el Array esté ordenado.

4. **De Burbuja:** los elementos adyacentes se comparan y se intercambian si están en el orden incorrecto. Se repite hasta que no haya más intercambios necesarios, es decir, el Array está ordenado.

```
void ordenamientoBurbuja(int array[], int SIZE) {
    for (int i = 0; i < SIZE - 1; i++) {
        for (int j = 0; j < SIZE - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j]; // Intercambio de elementos
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
```

```
}
```

El bucle `for` externo recorre en cada pasada el array desde `i = 0` hasta `i < SIZE - 1`.

El bucle `for` interno recorre el array desde `j = 0` hasta `j < SIZE - i - 1`. Escribimos `SIZE - i - 1` debido a que en cada pasada del bucle externo, los elementos más grandes ya se encuentran en la posición correcta al final del Array, por lo que no es necesario volver a compararlos.

El bucle `for` interno compara pares de elementos adyacentes: `array[j]` y `array [j+1]`. Si el primero es mayor que el segundo, significa un mal orden y se realiza un intercambio de elementos. Esta se realiza a través de la variable `temp` (espacio temporal que permite intercambiar los valores sin perder el valor original de uno de los elementos). Los elementos más grandes van sucesivamente colocandose al final del array, repitiendo el proceso hasta que no se realicen más intercambios. Es decir, el array está ordenado.

5. **Insertar Ordenado:** inserta los elementos al Array manteniendo el orden ascendente.

```
void insertarOrdenado(int array[], int SIZE, int element) {
  int i = SIZE - 1;

  while (i >= 0 && array[i] > element) { // Desplazar elementos mayores
                                            hacia la derecha
       array[i + 1] = array[i];
       i--;
  }

   array[i + 1] = element;  // Insertar el elemento en su posición correcta
}

int main() {
  int array[10];

  for (int i = 0; i < 10; i++) {
       int num;
       cin >> num;
       insertarOrdenado(array,i,num);
       mostrarArray(array,10);
  }
return 1;
}
```

Tenemos el array, su tamaño `SIZE` y un elemento `element`. Se declara y define una variable `i = SIZE - 1`, que representa el índice del último elemento del arreglo.

El bucle `while` desplaza elementos mayores hacia la derecha para hacer espacio al `element` en la parte ordenada del array. Se ejecuta mientras `i >= 0` y el elemento en la posición `i` sea mayor a `element`. Se desplaza el elemento de la posición `i` a la derecha. Después de hacer los intercambios necesarios, se coloca `element` en su posición correcta.

Luego, en la función `main` *se declara un array de tamaño 10 y se utiliza un bucle* `for` *para solicitar al usuario ingresar 10 números, se los ordena con la función que acabamos de crear y se imprime en pantalla el array resultante después de cada intercambio.*

Estructuras de Almacenamiento Físico (Archivos)

El almacenamiento en memoria es más rápido porque no se mantiene de manera permanente. El almacenamiento físico, por otro lado, guarda la información en archivos en el disco. Se mantiene permanentemente, y es más lento.

Archivos de Texto y Binarios

Archivos de texto: almacenan información en forma de carácteres legibles (ASCII o Unicode).

Archivos Binarios: almacenan datos en binario (secuencia de bits). Permite almacenar cualquier tipo de datos (números, imágenes, videos o estructuras complejas). Lo utilizaremos para guardar registros.

	Archivos de Texto	Archivos Binarios
Ventajas	Legibles por humanos	Almacenan datos estructurados y complejos
	Fáciles de editar con editores de texto	Manipulación eficiente de datos
	Ocupan menos espacio	Procesamiento más rápido
Desventajas	Limitaciones en la representación de datos	No legibles por humanos
	Menos eficientes en tiempo de lectura/escritura	Requieren programas específicos para interpretación y manipulación

Crear y Abrir un Archivo: Para abrir un archivo utilizamos la función `fopen`. Hay que pasarle dos parámetros: el nombre del archivo y el modo de apertura. Pueden combinarse entre ellos para generar “rb” o “wb”. Por ej:

```
FILE* archivo = fopen("estudiantes_binarios.dat", "wb")
```

Modos de Apertura:

→ `"r"`: modo lectura. El archivo debe existir previamente. El puntero de posición se coloca al principio del archivo.
→ `"w"`: modo escritura. Si el archivo no existe, se crea. Si el archivo ya existe, su contenido se elimina. El punto de posición se coloca al principio del archivo.
→ `"a"`: modo de adjuntar. Si el archivo no existe, se crea. Si el archivo ya existe, los nuevos datos se añaden al final. El puntero de posición se coloca al final del archivo.

→ `"b"`: modo binario. El archivo se lee o se escribe en su forma binaria original, sin realizar conversiones. Lo utilizamos para archivos binarios, como imágenes o archivos estructurados.
→ `"t"`: modo texto. Es el modo predeterminado si no se especifica "b". Los caracteres de nueva línea se pueden convertir automáticamente según la plataforma.
→ `"+"`: permite tanto lectura como escritura.

→ `"rb"` : Apertura en modo de lectura binaria.
→ `"rb+"` : lectura y escritura binaria de un archivo existente sin borrar su contenido.
→ `"wb"` : Apertura en modo de escritura binaria.
→ `"ab"` : Apertura en modo de agregado binario.
→ `"r+"` : Apertura en modo de lectura/escritura. El archivo debe existir.
→ `"w+"` : Apertura en modo de lectura/escritura. Si el archivo existe, su contenido se trunca. Si no existe, se crea uno nuevo.
→ `"a+"` : Apertura en modo de lectura/escritura. Los datos se escriben al final del archivo sin truncar su contenido existente. Si el archivo no existe, se crea uno nuevo.

Escribir un Archivo: utilizamos la función `fwrite.`

```
fwrite(lo_que_vamos_a_guardar, tamaño, cantidad, variable_archivo);
```

```
fwrite(estudiantes, sizeof(Estudiante), 3, archivo);
```

`tamaño`**:** en bytes de cada elemento a escribir.
`cantidad`**:** es el número de elementos que se quiere guardar en el archivo.

Ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;

struct Estudiante {
  int codigo;
  char nombre[50];
  int edad;
};

Estudiante dameEstudiante(int codigo) { // su número de índice en el array
  Estudiante estudiante;
  estudiante.codigo = codigo;

  cout << "Ingrese el nombre del estudiante: ";
  cin.getline(estudiante.nombre, sizeof(estudiante.nombre));

  cout << "Ingrese la edad del estudiante: ";
  cin >> estudiante.edad;
  cin.ignore(); // Ignorar el salto de línea pendiente en el búfer
```

```
  return estudiante;
}

int main() {
  Estudiante estudiantes[3];

  for (int i = 0; i < 3; i++) {
       cout << "Ingrese los datos del estudiante " << i + 1 << ":" << endl;
       estudiantes[i] = dameEstudiante(i + 1);
  }

   FILE* archivo = fopen("estudiantes_binarios.dat", "wb");
    if (archivo != NULL) { //si se pudo abrir el archivo correctamente
       fwrite(estudiantes, sizeof(Estudiante), 3, archivo);
       fclose(archivo);
       cout << "Archivo creado exitosamente." << endl;

   } else {
        cout << "No se pudo crear el archivo." << endl;
     }
return 0;
}
```

Leer un Archivo: utilizamos la función `fread`.

```
fread(&variable_donde_guardamos, tamaño, cantidad, variable_archivo);
```

```
fread(&estudiante, sizeof(Estudiante), 1, archivo)
```

Ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;

struct Estudiante {
  int codigo;
  char nombre[50];
  int edad;
  };

int mostrarEstudiantes() {
  FILE* archivo = fopen("estudiantes_binarios.dat", "rb");
  if (archivo != NULL) { //si se pudo abrir el archivo correctamente
       Estudiante estudiante;
       while (fread(&estudiante, sizeof(Estudiante), 1, archivo) == 1) {
  // si lee con éxito devolverá 1, sigue el bucle
            cout << "Estudiante encontrado:" << endl;
            cout << "Código: " << estudiante.codigo << endl;
```

```
            cout << "Nombre: " << estudiante.nombre << endl;
            cout << "Edad: " << estudiante.edad << endl;
        }
        fclose(archivo);
  } else {
  cout << "No se pudo abrir el archivo para lectura." << endl;
  }
return 0;
}
int main() {
  mostrarEstudiantes();
return 0;
}
```

Buscar en un Archivo

Secuencial:

```
#include <iostream>
#include <cstring>
using namespace std;

struct Estudiante {
  int codigo;
  char nombre[50];
  int edad;
  };

int buscarEstudiante(const char* nombreBuscado) {
  FILE* archivo = fopen("estudiantes_binarios.dat", "rb");
    if (archivo != NULL) { //si se pudo abrir el archivo correctamente
       Estudiante estudiante;
       while (fread(&estudiante, sizeof(Estudiante), 1, archivo) == 1) {
           if (strcmp(estudiante.nombre, nombreBuscado) == 0) {
           // compara las dos variables y si son iguales devuelve 0
                cout << "Estudiante encontrado:" << endl;
                cout << "Código: " << estudiante.codigo << endl;
                cout << "Nombre: " << estudiante.nombre << endl;
                cout << "Edad: " << estudiante.edad << endl;
                fclose(archivo);
           return 1;
           }
    }
    fclose(archivo);
    } else {
       cout << "No se pudo abrir el archivo para lectura." << endl;
    }
return 0;
}
```

```
int main() {
  char nombre[50];
  cout << "Ingrese el nombre del estudiante a buscar: ";
  cin >> nombre;

  if (!buscarEstudiante(nombre)) {
       cout << "Estudiante no encontrado." << endl;
  }
return 0;
}
```

Directa: utilizamos la función `fseek`.

```
fseek(variable_archivo, offset, origen);
```

```
fseek(archivo, (codigoBuscado - 1) * sizeof(Estudiante), SEEK_SET);
```

`variable_archivo:` puntero al archivo en el que se realizará la búsqueda.

`offset:` número de bytes a desplazarse, relativo a la posición indicada por origen.

`origen:` valor inicial del desplazamiento del puntero. Puede ser: `SEEK_SET` (desde el principio del archivo), `SEEK_CUR` (desde la posición actual del puntero), `SEEK_END` (desde el final del archivo).

Ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;

struct Estudiante {
  int codigo;
  char nombre[50];
  int edad;
  };

int buscarEstudiante(int codigoBuscado) {
  FILE* archivo = fopen("estudiantes_binarios.dat", "rb");
    if (archivo != NULL) { //si se pudo abrir el archivo correctamente
       Estudiante estudiante;
       fseek(archivo, (codigoBuscado -1) * sizeof(Estudiante),SEEK_SET);
       if(fread(&estudiante,sizeof(Estudiante), 1, archivo= ==1) {
            cout << "Estudiante encontrado:" << endl;
            cout << "Código: " << estudiante.codigo << endl;
            cout << "Nombre: " << estudiante.nombre << endl;
            cout << "Edad: " << estudiante.edad << endl;
            fclose(archivo);
           return 1;
    }
```

```
    fclose(archivo);
    } else {
       cout << "No se pudo abrir el archivo para lectura." << endl;
    }
return 0;
}

int main() {
  int codigo;
  cout << "Ingrese el código del estudiante a buscar: ";
  cin >> codigo;

  if (!buscarEstudiante(codigo)) {
       cout << "Estudiante no encontrado." << endl;
  }
return 0;
}
```

Editar un Archivo

```
#include <iostream>
#include <string>
#include <cstring>

struct Estudiante {
  int codigo;
  char nombre[50];
  int edad;
  };

void editarEdadEstudiante(int codigoBuscado, int nuevaEdad) {
  FILE* archivo = fopen("estudiantes_binarios.dat", "rb+");
    if (archivo != NULL) { //si se pudo abrir el archivo correctamente
       Estudiante estudiante;
       bool encontrado = false;
       while(!encontrado && fread(&estudiante,sizeof(Estudiante), 1,
archivo ==1) {
         if(estudiante.codigo == codigoBuscado) {
            encontrado = true;
            estudiante.edad = nuevaEdad;
            fseek(archivo, -sizeof(Estudiante), SEEK_CUR);
            fwrite(&estudiante, sizeof(Estudiante), 1, archivo);
            cout << "Edad actualizada exitosamente." << endl;
         }
       }
       if (!encontrado) {
          cout << "No se encontró un estudiante con ese código." << endl;
        }
       fclose(archivo);
    } else {
```

```
        cout << "No se pudo abrir el archivo para lectura y escritura." <<
endl;
    }
}
int main() {
  int codigoBuscado, nuevaEdad;
  cout << "Ingrese el código del estudiante a buscar: ";
  cin >> codigoBuscado;
  cout << "Ingrese la nueva edad: ";
  cin >> nuevaEdad;

   editarEdadEstudiante(codigoBuscado, nuevaEdad);
return 0;
}
```

Cargar frases:

```
#include <iostream>
using namespace std;

int main() {
  // Forma 1
  char nombre[30];
  cin.getline(nombre, sizeof(nombre));

  // Forma 2
  string nombre2;
  getline(cin,nombre2);

  cout << nombre;
  cout << nombre2;
  return 0;
}
```

Fechas: hacer de fecha un entero y multiplicar por múltiplos de 10 para calcular año, mes, día. Restar año, mes, día cuando sea necesario.

Mostrar todos los elementos del Array:

```
for (int i = 0; i < size; i++){
        cout << i << ". " << array[i] << endl;
   }
```

Datos:

→ Siempre cerrar el archivo.

→ Se pueden poner structs dentro de structs.

→ Poner la misma cantidad de elementos, porque a pesar de existir el carácter nulo, también está la posición cero del Array.