

## **Programación Estructurada:**

- **Estructuras de Control:** Para organizar el flujo de ejecución de las acciones del programa.
- **Estructuras de Programa:** Para organizar los módulos del programa.
- **Estructuras de Datos:** Para organizar los datos que se van a manejar en el programa.

## **Estructuras de Datos:**

- **Simples o Básicos:** Caracteres, reales, flotantes.
- **Estructurados:** Colección de valores relacionados. Se caracterizan por el tipo de dato de sus elementos, la forma de almacenamiento y la forma de su acceso.
  - **Estructuras Estáticas:** Su tamaño en memoria se mantiene inalterable durante la ejecución del programa, y ocupan posiciones fijas. Arreglos, cadenas, estructuras.
  - **Estructuras Dinámicas:** Su tamaño varía durante el programa y no ocupan posiciones fijas. Listas, pilas, colas, arboles, grafos.

## **Clasificaciones:**

- **Según donde se almacenan,**
  - **Internas:** Memoria principal.
  - **Externas:** Memoria auxiliar.
- **Según tipos de datos de sus componentes.**
  - **Homogéneas:** Todas del mismo tipo.
  - **No homogéneas:** Distintos tipos.
- **Según la implementación.**
  - **Provista por lenguajes:** Básica.
  - **Abstractas:** TDA – Tipo de dato abstracto que puede implementarse de diferentes formas.
- **Según la forma de almacenamiento.**
  - **Estáticas:** Ocupan posiciones fijas y su tamaño nunca varía durante todo el módulo.
  - **Dinámicas:** Su tamaño varía durante el módulo y sus posiciones también.

## **Tema #1 – Arreglos Unidimensionales.**

**Arreglos Unidimensionales:** Un conjunto de valores, todos del mismo tipo de dato, puede formar un grupo lógico o lista. Dicha lista, si contiene elementos individuales del mismo tipo de llama *Unidimensional*. Entonces, un arreglo unidimensional es una lista de valores relacionados con el mismo tipo de dato que se almacena usando un nombre de grupo único.

- Secuencia en memoria de un número determinado de datos del mismo tipo.
  - Los datos se llaman **elementos** del arreglo y se numeran consecutivamente. Dichos números se denominan **índice o subíndice** del arreglo. Localizan la posición.
  - Los elementos del arreglo se almacenan siempre en **posiciones consecutivas**,
  - Los elementos **se acceden en forma directa**, indicando el nombre del arreglo y el subíndice. *Variable Indizadas.*
- 
- **Nombre del Arreglo:** Todos los elementos del arreglo tienen el mismo nombre.
  - Los **índices** de arreglos siempre tienen límite inferior, **indexación basada en cero**, y como límite superior *tamaño físico -1*.
  - Para **referirse a un elemento** hay que especificar:
    - Nombre del arreglo == C
    - Número de posición == C[0]

**Declaración:** `TipoDeDatos NombreArreglo [Tamaño]`

- El **tamaño de un arreglo** puede ser un valor constante o estar representado por una constante.
- Se puede definir un **tipo de tabla unidimensional**
  - **TYPEDEF float Arreglo [].**
  - **Arreglo NombreArreglo.**

## Partes de un Arreglo:

- **Dirección:** Ubicación en memoria.
- **Tipo base:** Determina cuánta memoria ocupa cada variable.
- **Tamaño:** Cantidad de elementos del arreglo

**C++ NO COMPRUEBA QUE LOS ÍNDICES ESTEN DENTRO DEL RANGO DEFINIDO.**

**Referencia e Inicialización:** Se los puede referenciar por medio del subíndice, utilizando expresiones con resultado entero.

## Cargar un Arreglo:

- Puede **cargarse en forma completa** al declarar la variable (inicializar).
- Puede **cargarse elemento a elemento**.
- Si se **cargan menos elementos** el resto se inicializa en cero.
- Si hay una recurrencia, pueden **cargarse con un ciclo**.
- Puede **leerse desde el teclado**.
- Puede **cargarse con información que se va produciendo durante el proceso**.

## Almacenamiento:

- Los elementos de los arreglos se **almacenan en bloques contiguos**.
- **Espacio ocupado** por arreglo es la suma de cada elemento con un tamaño de 4 bytes.
- **Direcciones:**
  - & Arreglo [índice] = 1000 = D
  - & Arreglo [índice] = D + índice \* tamaño.

**Acceso Directo:** La variable A se refiere a un espacio de la memoria donde están almacenados en forma contigua los elementos. El contenido es la dirección donde comienza a almacenarse el arreglo, que también es la dirección del primer elemento del arreglo. El acceso a cualquier elemento del arreglo tiene la misma complejidad ya que cuando se lo referencia se hace un cálculo y se lo extrae de la dirección correspondiente

## Arreglo Lineales: Tamaño Físico & Real.

- Puede ocurrir que al compilar se conozca el **tamaño máximo** que puede tener un arreglo, pero no el **tamaño real**, el que se conocerá en tiempo de ejecución y que incluso puede variar.
- Es necesario **definir una variable** que contenga el **tamaño actual del arreglo**.
- **Tamaño Físico:** Cantidad *máxima* de elementos que tiene un arreglo.
- **Tamaño Real:** Cantidad *exacta* de valores dentro de un arreglo.
  - También se lo conoce como **tamaño actual o lógico**.
  - Puede ser **menor o igual al tamaño físico**, pero **nunca mayor**

**Borrar Elemento:** Implica correr los elementos que están a su derecha una posición hacia la izquierda, comenzando por el más cercano al elemento a borrar. Y, además, decrementar en uno la cantidad real.

**Borrar Rango:** Se debe borrar desde Posición-Uno hasta Posición-Dos. Por lo tanto, los elementos siguientes a Posición-Dos, hasta el final del arreglo deberán correrse (**Posición Dos – Posición Uno + 1**) posiciones a la izquierda.

**Insertar Elemento:** Implica correr los elementos una posición hacia la derecha, a partir de la posición en la que se va a insertar. Y además, incrementar en uno la cantidad real.

**Buscar Elemento:** El algoritmo difiere si el arreglo está o no ordenado.

- Si **no hay orden** deben examinarse los elementos de izquierda a derecha hasta encontrar el elemento buscado o hasta que se hayan examinado todos los elementos. **Búsqueda Secuencial**.
- Si **hay orden** debe examinarse el elemento central del arreglo, si el elemento a buscar es menor que éste, se buscará en la primera mitad, sino se buscará en la segunda mitad. Se repite esta acción hasta que se encuentre el elemento o se obtenga una mitad consistente de un solo elemento que no es el buscado. **Búsqueda Binaria**.

**Pasaje de Arreglos a Funciones:** El parámetro formal *int a []* es un **parámetro de arreglo**.

- Los corchetes sin expresión de índice adentro son los que se usan para indicar un parámetro de arreglo.
- Un parámetro de arreglo no efectúa una copia del contenido del arreglo, sino que copia la referencia a la dirección de memoria en la que empieza el arreglo.

- Cuando usamos un arreglo como argumento en una llamada de función, cualquier acción que se efectúe con el parámetro de arreglo se efectúa con el argumento arreglo, así que la función puede modificar los valores de las variables indizadas del argumento arreglo.

**Parámetro de Arreglo Constante en Funciones:** Cuando usamos un argumento arreglo en una llamada de función, la función puede modificar los valores almacenados en el arreglo. Podemos indicarle a la computadora que no pensamos modificar el argumento arreglo, y entonces la computadora se asegurará de que el código no modifique inadvertidamente algún valor del arreglo.

- Un parámetro de arreglo modificado con un **constante** es un **parámetro de arreglo constante**
  - Void (**const** int a [], int Tamaño);

## Tema #2 – Arreglos Multidimensionales.

- Son arreglos que permiten varios índices.
- A los arreglos **bidimensionales** se los llama comúnmente matrices, tablas o arreglos de arreglos.
- A los de **tres dimensiones** se los denomina cubos.
- Y al resto, en general, se los denomina multidimensionales.

### Declaración:

- Formato de declaración (Bidimensionales):
  - **TipoDatos NombreArreglo [Filas][Columnas]**
  - **Typedef TipoDatos NombreArreglo [][]**

### Inicialización:

- Asignaciones. Matriz [10][10].
- Enumerando elementos. Matriz [][] = {{1, 2, 3}, {4, 5, 6}}.
- Se pueden definir arreglos con componentes de otros arreglos previamente definidos.
- Se pueden definir tablas como *constants*, de la misma manera que se hace con los tipos elementales.
- Se puede omitir la información del primer concreto, después es necesaria en todos los demás.

## Tema #3: Ordenamiento y Mezcla Arreglos Unidimensionales.

### Operaciones con Arreglos:

- **Búsqueda:** Secuencial, Binaria.
- **Mezcla & Intercalación:** De dos arreglos ordenados.
- **Ordenamiento:** Algoritmos básicos y avanzados.

### Concepto de *Ordenación*:

- **Ordenar:** Significa reagrupar o reorganizar un conjunto de datos en algún determinado orden con respecto a uno de los campos del conjunto. El campo por el cual se ordena un conjunto de datos se denomina *clave*.
- Formalmente, se define la **ordenación** de la siguiente manera: Sea A una lista de N elementos: A1, A2, ..., An.
- **Ordenar** significa *permutar* estos elementos de tal forma que los mismos queden de acuerdo con un orden preestablecido.
- **Ordenación Interna:** Los datos se encuentran almacenados *en memoria* y son de acceso aleatorio o directo.
- **Ordenación Externa:** Los datos están *en un dispositivo de almacenamiento externo* y *es probable que no tengan acceso directo*. Su ordenación es más lenta que la interna.

### Métodos Simples:

- **Selección Directa:** Se debe encontrar el elemento más pequeño del arreglo e intercambiárselo por el elemento de la primera posición, luego encontrar el segundo más pequeño e intercambiárselo por el elemento de la segunda posición y continuar de esta manera hasta que se haya ordenado el arreglo entero.
  - **Análisis →** Este es uno de los algoritmos más simples de ordenamiento. Cada elemento se mueve como máximo una vez. El número de intercambios realizados es menor que en el *Burbuja*. Este algoritmo realiza muchas menos operaciones *intercambio* que el de la burbuja. Una desventaja respecto al *Burbuja Centinela* es

que no mejora su rendimiento cuando los datos ya están ordenados o parcialmente ordenados. Este método es recomendable para un número pequeño de elementos.

- **Inserción Directa:** En este algoritmo consideramos un elemento a la vez y lo movemos hasta el lugar correcto, entre aquellos que ya han sido considerados. El elemento se inserta moviendo los elementos superiores a él una posición a la derecha y ocupando la posición vacante.
  - **Análisis** → En el **mejor de los casos** el arreglo estará inicialmente ordenado, entonces el bucle interno sólo ejecuta el paso de comparación que siempre será falso. Esto implica que en el mejor de los casos el número de comparaciones será **n-1**. El **peor de los casos** el arreglo está inversamente ordenado y el número de comparaciones a realizar será el máximo. En el **caso promedio** los elementos aparecen en el arreglo en forma aleatoria, y puede ser calculado mediante la *suma del mejor y peor caso dividido entre dos*. A pesar de ser un método ineficiente y recomendable solo para un número pequeño de elementos, es intuitivo y de muy fácil implementación
- **Burbuja:** Lleva los elementos más pequeños hacia la parte izquierda del arreglo. Realiza repetidamente el **intercambio de pares de elementos adyacentes** hasta que estén todos ordenados. Se hacen **varias pasadas** a través del arreglo. En cada pasada se comparan pares de elementos. Si están en orden creciente, se los deja como está, sino se los intercambia.
- **Burbuja Mejorada V2:** Es una versión mejorada del algoritmo anterior, donde se realiza una comparación menos por cada pasada, aprovechando el hecho de que los elementos en el extremo derecho van quedando ordenados y no tiene sentido volver a compararlos.
- **Burbuja Mejorada c/Centinela:** El algoritmo finaliza cuando no se produce intercambio alguno entre los elementos del arreglo, pues esto indica que ya quedó. Este algoritmo utiliza una marca o señal para indicar que no se ha producido ningún intercambio en una pasada. Al final de cada pasada se evalúa si no se han hecho intercambios y en ese caso se termina el ciclo externo.
  - **Análisis** → Si el **arreglo está completamente desordenado**, peor caso, el ciclo (while) se comporta como el ciclo (for) de las versiones anteriores, y la cantidad de comparaciones e intercambios realizados es similar. Si el **arreglo queda completamente ordenado** en alguna pasada intermedia, nos ahorraremos todas las comparaciones de las restantes. Si el **arreglo está inicialmente ordenado**, mejor caso, sólo se ejecuta una pasada sobre el mismo, realizando **n-1** comparaciones y ningún intercambio.

#### Método Avanzado:

- **Merge Sort:** En este método se unen dos estructuras ordenadas para formar una sola ordenada correctamente. Consiste en dividir en dos partes iguales el vector a ordenar, ordenar por separado acá una de las partes, y luego mezclar ambas partes, manteniendo el orden, en un solo vector ordenado. Tiene la ventaja de tener la complejidad logarítmica **n log (n)**. Su desventaja radica en que se requiere de un espacio extra para el procedimiento.

#### Comportamiento de los Métodos de Ordenación: Criterios:

- **Estabilidad:** Es *estable* si valores iguales en el arreglo guardan su orden relativo.
- **Naturalidad:** Es *natural* si se tiene en cuenta la posible ordenación del arreglo, aunque sea parcial.
- **Orden Definitivo:** Es aquel que, en cada iteración, pone un elemento en su sitio definitivo, es decir la parte ordenada del arreglo ya es definitiva. Sólo sirve para el caso de ordenar los N° primeros elementos.
- **Eficiencia.**

### Tema #4: Complejidad Análisis de Algoritmos.

- Desde el **punto de vista computacional**, es necesario disponer de alguna forma de comparar una solución algorítmica con otra, para conocer cómo se comportarán cuando las implementemos, especialmente al atacar problemas grandes.
- La **complejidad algorítmica** es una métrica teórica que se aplica a algoritmos en este sentido.
- Es un concepto fundamental para todos los programadores.
- Entender la complejidad es importante porque para resolver muchos problemas, utilizamos algoritmos ya diseñados. Conocer el valor de su complejidad puede ayudarnos a escoger uno u otro.

#### Características de los Algoritmos.

- **Eficaces:** Cumplen con el requerimiento solicitado.
- **Eficientes:** Lo hacen mejor que otros.
  - **Rápido:** Eficiencia temporal.
  - **Buen uso de Recursos:** Eficiencia espacial
- **Legibles:** Claros y bien estructurados.
- **Generales:** Capaces de resolver una clase de problemas lo más amplia posible, de uso y mantenimiento fácil.

- Al **tiempo** que consume un algoritmo para resolver un problema lo llamamos **complejidad temporal**.
- A la **memoria** que utiliza el algoritmo la llamamos **complejidad espacial**.
- La complejidad espacial, en general, tiene mucho menos interés. **El tiempo es un recurso mucho más valioso que el espacio.**
- Así que cuando hablamos de *complejidad* a secas, nos estamos refiriendo a **complejidad temporal**.

### Complejidad:

- **¿Qué medimos?**
  - Tiempo de ejecución
- **¿De qué depende?**
  - Técnica utilizada para resolver el problema.
  - Tamaño de la entrada.
  - Otros factores (HW y SW).
    - Características de la máquina.
    - Software utilizado.
- **¿Cómo medirlo?**
  - Experimentalmente → a posteriori.
  - Estimándolo matemáticamente → a priori.

### Eficiencia de los Algoritmos: Tiempo de ejecución:

- El tiempo de ejecución de un algoritmo típicamente crece con el tamaño de la entrada.
- Pero no solo depende del tamaño, sino que influye el contenido de los datos.
- Se distinguen tres casos.
  - **Mejor caso:** Mínimo tiempo posible.
  - **Peor caso:** Máximo tiempo posible. Es el caso más representativo.
  - **Caso medio:** Tiempo promedio.

**La mejor opción es analizar el peor caso.**

### Análisis Teórico:

- Utiliza una **descripción de alto nivel del algoritmo**, en lugar de una implementación.
- Caracteriza los tiempos de ejecución como **una función** del número de elementos que deben ser procesados →  $T(N)$ .
- Toma en cuenta **todas las posibles entradas** y se emplea el **peor caso**.
- Permite evaluar el tiempo de ejecución de un algoritmo **independientemente** del entorno del hardware y software.
- Interesa la **velocidad de crecimiento** del tiempo de ejecución en función del tamaño de la entrada. Comportamiento asintótico.

### Cálculo de $T(N)$ .

- **Cálculo estimado de  $T(N)$ :** Se mide el tiempo de ejecución empíricamente. Requiere trabajo planificado y sistemático, presentación de los resultados y validez.
- **Cálculo de la expresión de  $T(N)$ :** Se realiza un análisis matemático del algoritmo, ya sea para calcular una expresión de  $T(N)$  o una cota superior a la misma. Expresión asintótica, orden de magnitud.
- Para la descripción del algoritmo se emplea un **pseudocódigo**.
  - Tiene menos nivel de detalle que un programa.
- Los pseudocódigos indican:
  - Control de flujo.
  - Declaraciones de métodos.
  - Llamadas a métodos.
  - Retorno de valores.
  - Expresiones.
- Los bucles son el término dominante para determinar la eficiencia.
  
- El tiempo se mide en función de cada operación o instrucción elemental. *Primitiva*.
  - Asignación.
  - Llamada a método.
  - Operación aritmética, lógica, relacional.
  - Indexación en array.

- Seguir una referencia.
- Volver de un método.
- Las operaciones primitivas son **independientes** de la máquina, del lenguaje, del compilador y de cualquier otro elemento del hardware o software.
- Cada una de ellas **se contabiliza como una operación elemental**.
- Se asume que tiene un tiempo de ejecución constante en un determinado modelo RAM. *Costo Unitario*.

### Complejidad: Función de Orden:

- Expresar  $T(N)$  como el **número de expresiones elementales (primitivas)** ejecutadas por el algoritmo.
- **Expresar  $T(N)$  asintóticamente:** Interesa el orden de magnitud de la función, valores grandes de  $n$ . El objetivo es ver cómo crece el tiempo de ejecución cuando crece el tamaño de la instancia. Se prescinde de las constantes multiplicativas y de los órdenes de magnitud menores.

### Notación O:

- Interesa conocer el comportamiento del tiempo de ejecución cuando la cantidad de datos ( $N$ ) crece.
- $F(n)$  es  $O(G(n)) \rightarrow$  Orden de Complejidad de  $G(n) \rightarrow F(n)$  menor/igual  $C * G(n)$ .

### Propiedades:

- $C * O(F(n)) == O(F(n))$ .
- $O(F(n)) + O(G(n)) == O(F(n) + G(n))$ .
- $\text{Máximo}(O(F(n)), O(G(n))) == O(\text{Máximo}(F(n), G(n)))$ .
- $O(F(n)) * O(G(n)) == O(F(n) * G(n))$ .

### Jerarquía de Cotas:

- Un algoritmo de coste **constante** ejecuta un número constante de instrucciones. Un algoritmo que soluciona un problema en tiempo constante es lo ideal.
- El coste de un algoritmo **logarítmico** crece muy lentamente conforme crece  $N$ .
- Un algoritmo cuyo coste es **raíz N**, crece a un ritmo superior que otro que es logarítmico, pero no llega a presentar un crecimiento lineal. Cuando la talla se multiplica por 4, el coste se multiplica por dos.
- Un algoritmo cuyo coste temporal es **(n log n)** presenta un crecimiento del coste ligeramente superior al de un algoritmo lineal.
- Un algoritmo de coste **cuadrático** empieza a dejar de ser útil para las tallas medias o grandes, pues duplicar el tamaño del problema requiere 4 veces más tiempo.
- Un algoritmo de coste **cúbico** solo es útil para problemas pequeños, duplicar el tamaño del problema hace que se tarde 8 veces más tiempo.
- Un algoritmo de coste **exponencial** raramente es útil. Duplicar el tamaño del problema requiere un aproximado de 1000 veces más tiempo.

### Análisis de Complejidades:

- Búsqueda Lineal:  $O(N)$ .
- Búsqueda Binaria:  $O(\log_2 N)$ .
- Mezcla de Arreglos –  $N_1$  y  $N_2$  Elementos:  $O(N_1 + N_2)$ .
- Ordenamiento por Selección:  $O(N^2)$ .
- Ordenamiento por Inserción:  $O(N^2)$ .
- Ordenamiento por Burbuja:  $O(N^2)$ .
- Ordenamiento por Burbuja Mejorada:  $O(N^2)$ .
- Ordenamiento por Merge Sort:  $O(N \log N)$ .

**La complejidad es del algoritmo.**

**No perder tiempo intentando optimizar el código. Sino optimizar el algoritmo.**

## Tema #5: Cadenas.

### Cadenas de Caracteres:

- En C++, una cadena es un tipo de dato compuesto, un **arreglo de caracteres** que siempre incluye un 0 binario, llamado **terminador nulo (\0)** como elemento final del arreglo.
- **Caracteres:** Constante de carácter:
  - Un valor entero representado como un carácter entre comillas simples
- **Cadenas:** Series de caracteres tratados como una unidad:
  - Pueden incluir letras, dígitos y caracteres especiales.
  - Representadas entre comillas dobles.
  - Terminan siempre en un carácter nulo.
  - Pueden ser también llamadas strings.
- **Declaración:**
  - Se declaran como un arreglo de caracteres.
  - También se puede declarar como una variable de tipo **char \*** - puntero.
  - El tamaño de la cadena debe incluir un byte más para poder almacenar el carácter nulo.
  - Al asignar 'a', 'b', 'c', ... etcétera, no se coloca el terminador nulo. Imposibilita el uso de librerías.

### Entrada de Cadenas: >>

- Elimina los espacios en blanco que hubiera al principio.
- Lee dicha entrada hasta encontrar algún carácter de espacio en blanco, este permanecerá en el buffer

<b>Funciones en &lt;cstdio&gt;</b>	
<b>Prototipo de Función</b>	<b>Descripción</b>
int getchar (void)	Lee el próximo carácter desde la entrada estándar y lo retorna como un entero
char *gets (char*s)	Lee caracteres desde la entrada estándar en el arreglo S hasta un carácter de newline o end-of-file. Se agrega al array un carácter de terminación NULL
int putchar (int c)	Imprime el carácter almacenado en c
int puts (const char*s)	Imprime el string s seguido por un carácter de nueva línea
gets (c)	Lee una cadena con espacios
getchar (c)	Permite leer solo un carácter
<b>Funciones en &lt;cstring&gt;</b>	
strcpy (Dest, Orig)	Copia el valor de cadena tipo C <i>Origen</i> hacia la variable de cadena tipo C <i>Destino</i>
strncpy (Dest, Orig, Lím)	Funciona de igual forma que la anterior, sólo que se copian cuando mucho <i>Límite</i> de caracteres
strcat (Dest, Orig)	Concatena el valor de cadena tipo C <i>Origen</i> con el final de la cadena tipo C que se encuentra en la variable tipo C <i>Destino</i> .
strncat (Dest, Orig, Lím)	Funciona de igual forma que la anterior, sólo que se copian cuando mucho <i>Límite</i> de caracteres
strlen (Cadena)	Devuelve un entero igual a la longitud de <i>Cadena</i> . No cuenta el terminador nulo
strcmp (Cad1, Cad2)	Devuelve 0 si <i>Cad1</i> y <i>Cad2</i> son iguales. Devuelve un valor menor a 0 si <i>Cad1 &lt; Cad2</i> . Devuelve un mayor a 0 si <i>Cad1 &gt; Cad2</i> .
strncmp (Cad1, Cad2, Lím)	Funciona de igual forma que la anterior, sólo que se copian cuando mucho <i>Límite</i> de caracteres.
<b>Funciones en &lt;cctype&gt;</b>	
int isalpha (charExp)	Devuelve un número distinto de cero si evalúa una letra
int isalnum (char Exp)	Devuelve un número distinto de cero si evalúa una letra y/o número
int isupper (charExp)	Devuelve un número distinto de cero si evalúa una letra mayúscula
int islower (charExp)	Devuelve un número distinto de cero si evalúa una letra minúscula
int isdigit (charExp)	Devuelve un número distinto de cero si evalúa un dígito
int isspace (charExp)	Devuelve un número distinto de cero si evalúa un espacio
int toupper (charExp)	Devuelve el equivalente en mayúsculas
int tolower (charExp)	Devuelve el equivalente en minúsculas

### Cadenas de Caracteres: String:

- Puede ser utilizado para definir constantes simbólicas, variables o parámetros formales.
- No es posible emplear datos de tipo string en archivos.
- Hacen crecer el espacio de almacenamiento para acomodarse a los cambios de tamaño de los datos de la cadena por encima de los límites de la memoria asignada inicialmente.
- **Entrada & Salida:**
  - La entrada/salida sigue el mismo esquema que la de los tipos predefinidos simples.

- Si la definición de una variable de tipo string no incluye una asignación de un valor inicial, dicha variable tendrá como valor **por defecto la cadena vacía**.
- **Problemas con GetLine ():**
  - Luego de introducir algún dato, en el buffer queda almacenado el carácter de fin de línea que se introdujo tras teclear la edad, ya que este no es leído por el operador. En la siguiente iteración la función leerá el carácter almacenado en el buffer. Allí se produce el error.
  - La solución a este problema es eliminar los caracteres de espacios en blanco del buffer de entrada. De esta forma, el buffer estará realmente vacío.
  - El manipulador ws en el flujo cin, que extrae todos los espacios en blanco hasta encontrar algún carácter distinto.
  - Va antes del getline
- **Otra forma de solucionar el problema:**
  - Es posible que interese que la cadena vacía sea una entrada válida en el programa. En este caso, es necesario que el buffer se encuentre vacío en el momento de realizar la operación de entrada. Para ello, eliminaremos los caracteres que pudiera contener el buffer (no únicamente espacios en blanco) después de la última operación de lectura de datos, usando la función cin.ignore () .
  - Cin.Ignore () elimina todos los caracteres del buffer de entrada en el flujo especificado, hasta que se haya eliminado el npumero de caracteres indicado en el primer argumento, o bien se haya eliminado el carácter indicado en el segundo.
  - Se coloca después del getline

#### Función <string>

Prototipo de Función	Descripción
cin>>ws getline (cin, Variable, Delimitador)	Lee y almacena en una variable string todos los caracteres del buffer de entrada. Además, permite especificar el delimitador, de manera opcional Cuando se utiliza luego de un << genera problemas, por ello hay que limpia el buffer. “cin >> ws; ”
Comparación	== , !=, <, >, <= , >= , ...
Concatenación	+ , += , =
size () // length ()	Se utiliza para obtener el largo de la cadena. Utiliza la nomenclatura del punto
at (índice)	Devuelve el carácter en la posición especificada y lanzará una excepción si se accede a una posición inexistente. Se utiliza la nomenclatura del punto
substr (Posición, Longitud)	Obtiene una nueva sub-cadena a partir de posición
swap (Cadena)	Permite intercambiar una cadena con otra. Además, utiliza la nomenclatura del punto.
find (Sub-Cadena)	Devuelve la posición de la primera ocurrencia de la sub-cadena o frase.
rfind (Sub-Cadena)	Devuelve la posición de la última ocurrencia de la sub-cadena o frase.
erase (Posición, Cantidad)	Elimina cantidad de caracteres, a partir de posición. Utiliza la nomenclatura del punto
insert (Posición, Cadena)	Inserta la cadena o frase a partir de la posición. Utiliza la nomenclatura del punto.

#### Tema #6: Estructuras.

##### Estructuras: Registros:

- Colección de variables relacionadas bajo un mismo nombre.
- Pueden contener variables de diferentes tipos de datos.
- Se usa habitualmente para definir registros que van a ser almacenados en archivos o en arreglos.
- Combina con punteros, pueden crear listas enlazadas, pilas, colas, árboles.

##### Definición:

- **Struct** introduce la definición para una estructura.
- Dicha estructura posee un nombre que se utiliza para declarar variables de ese tipo (Opcional).
- Puede contener distintos campos con diferentes tipos y valores.
- La definición **no reserva espacio en memoria**, solo crea un nuevo tipo de dato que puede ser usado para declarar variables de estructura.
- En el **ámbito de visibilidad** de los campos de una estructura se restringe a la propia definición del registro. Los campos de un registro pueden ser de cualquier tipo de dato, simple o compuesto.
- **Inicialización:**
  - Mediante listas inicializadores.
  - Mediante sentencias de asignación.
  - Accediendo a sus campos mediante el operador punto
- Se admiten registros o estructuras anidadas.
- **Operaciones válidas:**

- Acceder a los campos de una estructura.
- Asignar una estructura a otra estructura del mismo tipo.
- Usar el operador `sizeof` para determinar el tamaño de una estructura.
- Obtener la dirección de una estructura.
- **Acceso:**
  - Se puede acceder a los campos de una estructura utilizando el **operador punto (.)** o utilizando el **operador flecha (->)**
  - Por defecto, los structs se pasan por valor, es decir, una copia.

#### **Asignaciones:**

- Se permiten asignaciones con registros completos.
- Es posible asignar un valor del tipo registro, completo, a una variable o campo, siempre que sean del mismo tipo.
  - Los operadores de comparación no son válidos entre los valores del tipo de registro.

#### **Tema #7: TDA.**

#### **Clasificación:**

- Provistas por los lenguajes: Básicas.
  - Arreglo; Estructura; String; Archivo; etcétera.
- Abstractas: TDA – Tipo de dato abstracto que puede implementarse de diferentes formas.
  - Un TDA es un tipo de dato **definido por el programador** que se puede manipular de un modo **similar a los datos provistos por el lenguaje**.
  - Un TDA está formado por **un conjunto de valores válidos** de datos y **un conjunto de operaciones primitivas** que se pueden realizar sobre esos valores.
  - Los usuarios pueden **crear** variables con valores del conjunto válido y **operar** sobre esos valores.

#### **Definición:**

- Los TDA proporcionan un mecanismo adicional mediante el cual se realiza una **separación clara entre interfaz y la implementación del tipo de dato**.
- La implementación consta de:
  - **La representación:** elección de estructuras de datos.
  - **Las operaciones:** elección de los algoritmos.
- La interfaz del TDA se asocia con las operaciones y datos del TDA, y es visible al exterior.

#### **Estructura:**

- Una **struct** permite que un grupo de variables que tienen una cierta relación lógica sean tratadas como **un todo**.
- Tener en cuenta que el usuario define:
  - Cómo va a representar el nuevo tipo de datos que está creando.
  - Cuáles van a ser las operaciones primitivas.
  - El código de dichas operaciones.
- El programador que utilice el TDA debe conocer la representación y la interfaz. La implementación queda oculta para el programador cliente.

#### **Implementación en C++:**

- Una de las características de **C++ que permite implementar TDA** son los **archivos de inclusión o cabecera** que se utilizan para agrupar en ellos variables externas, declaraciones de datos comunes y prototipos de funciones.
- Estos archivos de cabecera se incluyen en los archivos que contienen la codificación de las funciones, archivos fuente y también en los archivos de código que hagan referencia a algún elemento del archivo de inclusión.
- **Al implementar un TDA en C++ se agrupa** en este archivo la representación de los datos y la interfaz del TDA.
- Luego, en los archivos de programas que vayan a utilizar el TDA se debe escribir el include correspondiente.
- **Algunas consideraciones:**
  - Los programas fuentes necesitan conocer los datos públicos, es decir su **representación** y las **funciones primitivas** que se tienen disponibles.
  - La implementación de las funciones primitivas **es transparente** para el programa que usa el TDA, si bien debe existir y ensamblarse, el programador no necesita conocer ese código.
  - El archivo fuente (cpp) que tiene el código de las funciones, puede cambiarse y solo será necesario recomilar, sin que esto afecte al programa que usa el TDA, siempre que la representación e interfaz se mantenga.

#### **El Rol de la Abstracción:**

- **La abstracción** ha sido clave en la programación como mecanismo para controlar problemas de mayor complejidad.
- Ante la dificultad de dominar en su totalidad los objetos complejos, **se ignoran los detalles no esenciales**, tratando en su lugar con el **modelo ideal** del objeto y centrándonos en el estudio de sus aspectos esenciales.
- La abstracción es la capacidad para **encapsular y aislar información** del diseño y ejecución.
- **En la historia del software** la abstracción ha sido clave:
  - Nombres nemotécnicos en lugar de representaciones binarias.
  - Macroinstrucciones.
  - Estructuras de control. *Abstracción de Control*.
  - Procedimientos y funciones. *Abstracción de Control*.
  - TDA. *Abstracción de Datos*.

#### Ventajas de los TDA:

- Permite una **mejor conceptualización y modelización** del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
- **Mejora la robustez** del sistema.
- **Mejora el rendimiento**.
- **Separa la implementación de la especificación**. Permite la modificación y mejora de la implementación sin afectar la interfaz pública.
- **Permite la extensibilidad del sistema**. Los componentes de software reusables son más fáciles de crear y mantener.

### Tema #8: Pilas & Colas.

#### Pilas:

- Estructura de datos en la cual **el acceso está limitado al elemento más recientemente insertado**.
- El último elemento añadido a la pila es colocado en la **cima**, donde es accedido directamente, mientras que los elementos que llevan más tiempo son más difíciles de acceder.
- Operaciones.
  - Insertar/Apilar. Agrega un elemento nuevo en el tope de la pila
  - Eliminar/Desapilar. Elimina un elemento del tope de la pila. Deben ser eliminados en el orden inverso al que fueron situados.
  - Buscar
- LIFO. Last In, First Out.
- Las pilas se pueden implementar guardando los elementos en un arreglo de longitud fija. Y utilizando una variable para mantener la posición del último elemento colocado.
- Una pila puede estar vacía o llena.
- Si un programa intenta sacar un elemento de una pila vacía, se producirá un error por **desbordamiento negativo**.
- Si un programa intenta poner un elemento en una pila llena se producirá un error por **desbordamiento**

#### Colas:

- Estructura de datos en la cual se puede **eliminar el elemento más antiguo** y solo se puede **insertar después del más reciente**.
- Se puede identificar un frente y un final.
- Operaciones.
  - Insertar/Encolar. Insertar un elemento al final de la cola.
  - Eliminar/Desencolar. Eliminar un elemento del frente de la cola
- FIFO. First In, First Out.
- Las colas se pueden implementar guardando los elementos en un arreglo de longitud fija y utilizando dos marcadores para mantener las posiciones de frente y final de cola.
- Una cola puede estar vacía o llena.
- **Colas en Arreglos**.
  - Es la forma más eficiente de almacenar una cola.
  - En las **colas circulares** se une el extremo final con el extremo de la cabeza. De esta manera, la totalidad de las posiciones del array se utilizan para poder almacenar elementos de la cola **sin necesidad de desplazar elementos**.
  - Se implementan en un arreglo lineal. Se utilizan dos marcadores para mantener las posiciones de frente y final de la cola.

### Tema #9: Archivos.

## Introducción:

- Un archivo es una colección de datos almacenados juntos bajo un nombre común.
- Los archivos **almacenan información de manera permanente** en dispositivos de almacenamiento de memoria secundaria.
  - La información puede ser tanto programas, como datos que serán utilizados por los programas.
- Los archivos de datos pueden ser creados, leídos y actualizados por programas en C++.

## Archivos:

- Para tratar con un fichero se utilizan operaciones básicas que son:
  - Abrir.
  - Cerrar.
  - Escribir.
  - Leer.
- Los ficheros pueden ser **de lectura**, en cuyo caso diremos que el tipo de acceso es de entrada, o bien **de escritura**, con lo cual tendremos un fichero de salida.
- Los ficheros de salida pueden:
  - Crearse.
  - Actualizarse.
- También hay archivos de entrada/salida.
- El tipo de acceso determinará las **operaciones disponibles**.

## <fstream >

- Esta biblioteca nos provee de objetos que serán de entrada o salida.
  - Entrada = ifstream.
  - Salida = ofstream.
- **Información relativa a un Archivo/Fichero:**
  - Los objetos del tipo fichero son *objetos lógicos* que representan un fichero físico.
  - Para poder identificar de forma única el fichero físico que manejan, se necesita un nombre de fichero.

## Tipos de Archivos:

- Texto.
  - La información se almacena como una secuencia de caracteres, cada carácter individual se almacena utilizando una codificación estándar.
  - Al tratarse de un formato estandarizado, otros programas diferentes de aquel que creó el fichero podrán entender y procesar su contenido.
- Binarios.
  - La información se almacena con el mismo formato y codificación utilizado por el compilador para sus datos primarios.
  - Los números aparecen en su forma binaria.
  - Las cadenas conservan su forma ASCII.
  - La ventaja de los archivos binarios es su compatibilidad debido a que se usa menos espacio para almacenar más números usando su código binario que como valores de carácter individuales.

## Clasificación:

- **La dirección del flujo de los datos.**
  - Entrada: Ficheros cuyos datos se leen por parte del programa.
  - Salida: Ficheros que el programa escribe.
  - Entrada/Salida: Ficheros que se pueden leer y escribir.
- **Dependiendo de cómo se accede a los datos.**
  - Secuencial: El orden de acceso a los datos está determinado, del primero al último, de uno en uno.
  - Directo/Aleatorio: Se puede acceder de forma directa a un elemento concreto del fichero. El acceso es similar al de los arreglos.

## Flujo de E/S Asociados a Ficheros:

- Se dispone de un fichero de texto.
- Almacenado en memoria.
- Contiene información donde cada línea se encuentra terminada por un carácter, no visible, terminador de fin de línea.

- Aunque los datos almacenados en memoria se encuentran en formato binario, **son convertidos a su representación textual de ser escritos en el fichero.**
- Similarmente, cuando leemos del fichero de texto para almacenar la información en memoria **se produce una conversión de formato de texto a formato binario.**

#### **Entrada & Salida:**

- Un programa codificado en C++ realiza la entrada y salida de información a través de **flujos de entrada y salida** respectivamente.
- La entrada y salida de datos a través de los **flujos estándares** de entrada y salida, usualmente conectados con el teclado y la pantalla de la consola.
- Todo lo visto también **es aplicable** a los flujos de entrada y salida vinculados a **ficheros**.

#### **Tratamiento de Archivos – pág. 15**

#### **Lectura:**

- Una vez que el fichero está abierto, **podemos leer elementos con el operador de entrada >>**
- Se usa igual que en las instrucciones de entrada estándar, pero sustituyendo el flujo *cin* por el **nombre de la variable del tipo ifstream**.
- Dicho operador ignora los espacios en blanco

FicheroTexto.Unseft (ios :: skipws);	No ignora los espacios en blanco
FicheroTexto.Seft (ios :: skipws);	Ignora los espacios en blanco
Get (char & Ficher)	Lee un solo carácter y lo pone en la variable
Get (char* Fichero, int, Num, char Del)	Se usa para leer un número determinado de caracteres y termina si encuentra un salto de línea. El carácter delimitador es opcional y sirve para definir uno que, por defecto, es el salto de línea.
GetLine (char* Fichero, int Num, char Del)	Permite la lectura de toda una línea. Lee hasta que se encuentre el delimitador, haya leído Num -1 de caracteres o el final del fichero. Almacena el terminador nulo.

#### **Diferencias: Get & GetLine:**

- La función *get* () se detiene cuando ve el delimitador en el stream de entrada, pero no lo extrae del stream de entrada. Entonces, si se hace otro *get* () usando el mismo delimitador, retornará inmediatamente sin ninguna entrada contenida.
- La función *getline* (), por el contrario, sí extrae el delimitador del stream de entrada, pero no lo almacena en la cadena resultante.

#### **Archivos Binarios:**

- **istream & read (char\* S, streamsize N);**
  - Extrae N caracteres desde el archivo de entrada y los almacena en la cadena de caracteres S.
  - Para leer un tipo de dato que no sea una cadena de caracteres deberá hacer un *casting*.
  - Esta función copia un bloque de datos sin chequear su contenido.
  - Cuando no sepamos la medida en bytes de la información que queremos leer, deberemos usar la función de *sizeof == Fichero.read ((char\*) (& var), sizeof(double));*
- **ostream & write (const char\* S, streamsize N);**
  - Inserta en el archivo de salida N caracteres de la cadena de caracteres S.
  - Esta función tampoco valida contenido.

#### **Ficheros con Acceso Directo:**

- **Acceso Aleatorio (Directo).**
  - Cualquier carácter en el archivo abierto puede leerse en forma directa sin tener que leer primero en forma secuencial todos los caracteres almacenados antes que él.
- **Para proporcionar acceso aleatorio a los archivos** cada objeto *ifstream* crea en forma automática un marcador de posición de archivo *seekg/seekp*.
- Este marcador es un **numero entero largo** que representa un desplazamiento desde el principio de cada archivo e indica el lugar **desde donde se va a leer o a escribir el siguiente carácter**.

#### **Ficheros de Acceso: Seek:**

- Para leer y escribir ficheros de acceso directo usaremos las mismas operaciones que ya conocemos, la diferencia es que tenemos disponible la instrucción que nos **permite posicionarnos en el fichero**.
  - **Seek:** “Seek Get”
    - Se puede aplicar sobre ficheros de entrada, así como ficheros de E/S, e indica la posición del próximo get
- Para ficheros de salida, la posición para el próximo put se puede indicar con
  - **Seekp:** “Seek Put”
- La función **seek** nos permite acceder a cualquier posición del fichero, no tiene por qué ser exactamente al principio de un registro.
- Se hace referencia a la posición de carácter como su **desplazamiento desde el inicio** del archivo. Por lo tanto, el primer carácter tiene un desplazamiento de 0, y así...
- La resolución de la función **seek** es de 1 byte.
- Un desplazamiento positivo significa avanzar en el archivo y un desplazamiento negativo significa retroceder.
- Hay que tener en cuenta que el final de un fichero no es la posición del último elemento, sino la **posición del cursor después de leer el último elemento**.

#### Ficheros de Acceso: Tell:

- Para saber, en un momento determinado, **la posición en la que estamos**, esta se puede consultar con la instrucción **tell**.
  - **Tellg():** Devuelve la posición del cursor dentro del fichero de lectura.
  - **Tellp():** Devuelve la posición del cursor dentro del fichero de escritura.

## Tema #10: Punteros.

#### Punteros:

- Simulan por referencia.
- Estrecha relación con los arreglos y strings.
- Permiten el manejo dinámico de memoria.
- Permiten la implementación de estructuras de datos dinámicas.
- Permite implementar arreglos.
- Contienen direcciones de memoria como sus valores.
  - Las variables normales contienen un valor específico. **Referencia directa**.
  - Un **puntero** contiene la dirección de una variable que tiene un valor específico. **Referencia indirecta**.

#### Definición:

- Se usa \* con variables puntero.
  - **int \*Puntero.**
  - Define un *Puntero* a un int.
- Varios punteros requieren el de un \* antes de cada definición de variable.
  - **int \*Punt1, \*Punt2.**
- Se pueden definir punteros **a cualquier tipo de datos**.

#### Inicialización:

- Los punteros se inicializan a 0, NULL, o a una dirección de memoria concreta.
  - 0 o NULL indican que apuntan a nada.
  - Mejor NULL
  - **int \*Puntero = NULL;**

#### Operadores de Punteros:

- Operador dirección (**&**).
  - Retorna la dirección del operando.
  - **yPtr = &Variable.**
- Operador de indirección/desreferencia.
  - Retorna el contenido de **la variable cuya dirección está almacenada en el puntero o la variable a la que apunta** el puntero.
  - **\*yPtr** retorna 5 porque Variable = 5.
  - **yPtr = 7** entonces Variable = 7
  - El puntero indirección debe ser una variable a la izquierda del signo igual. No constante.

#### Expresiones y Aritmética:

- Sobre punteros se pueden realizar operaciones aritméticas.
  - Incrementar/decrementar un puntero.
  - Sumar/restar un entero a un puntero.
  - Esto implica que la dirección almacenada en la variable tipo puntero se incrementa/decremente tanto bytes como corresponde al tipo de dato apuntado.
  - Los punteros se pueden restar entre ellos.
- **Estas operaciones tienen significado cuando se apunta a un arreglo.**

**Tema #11: Estructuras Dinámicas.**