algoritmos primeros temas

**Algoritmos utiles saber como implementar**

**int exponente(int a,int b){ | int bin\_ent(int x){**

**if(b==0){ return 1;} | int aux=cifras(x);**

**else{ | if(aux==1){return x;}**

**if(b==1) {return a;} | return (x%10)+2\*bin\_ent(x/10);}**

**else { |**

**return a\*exponente(a,b-1);}}} |**

**int invertir(int x){ | int ent\_bin(int x){**

**int aux=cifras(x); | if(x==1){return 1;}**

**if(aux==1){return x;} | else{**

**return (x%10)\*exponente(10,aux-1)+invertir(x/10);}| return x%2+10\*ent\_bin(x/2);}}**

**void intercambiar(int&a,int&b){ int c=a; a=b; b=c;}**

**PALINDROMO**

la estrategia mas facil es invertir el numero y compararlo con el original

Sino vamos por el camino largo comparando de afuera hacia adentro cifras mas significativas con las menos significativas

**bool palindromoTedioso(int x){ //PALINDROMO ITERATIVO CAMINO LARGO**

**int digitos=cifras(x), divisor=exponente(10,digitos-1), i=1, aux=x;**

**bool pal=true;**

**while(pal and i<=digitos/2){**

**int primero=aux%10, ultimo=aux/divisor;**

**if(primero!=ultimo)return false;**

**i++;**

**aux=(aux%divisor)/10;**

**divisor/=100;}**

**return pal;}**

**bool capicua(int x){ //PALINDROMO RECURSIVO (capicua es un tipo de palindromo)**

**if(x<10){return true;}**

**else{**

**int primero=x%10;**

**int divisor=exponente(10,cifras(x)-1);**

**int ultimo=x/divisor;**

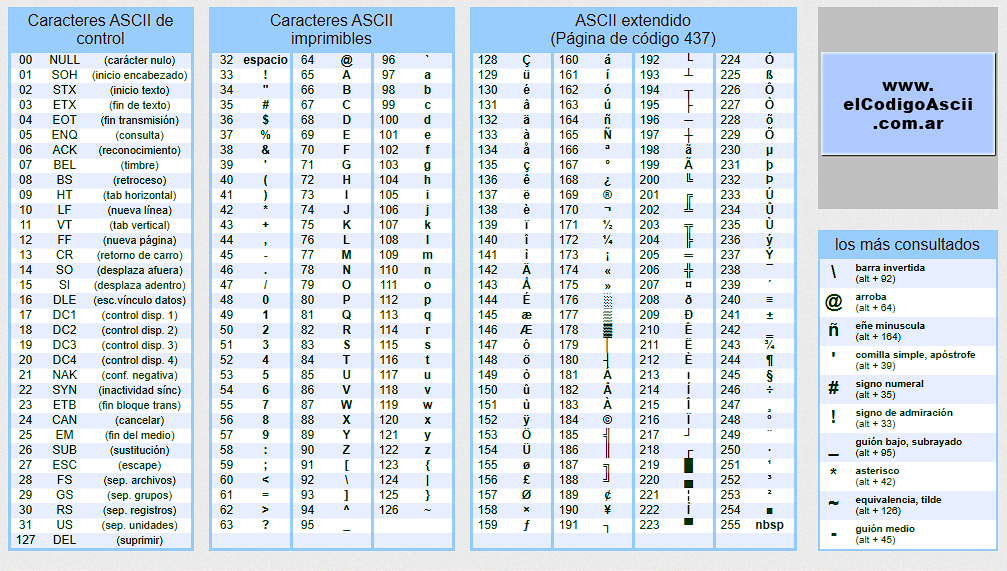
**cout<<x<<endl;**

**if(primero==ultimo){**

**return (true and capicua((x%divisor)/10));}**

**else{return false;}**

**}}**

****

* **la diferencia entre mayusculas y minusculas del codigo ascii es +-32**

**void mayusculas(char&c){c=char(toupper(c));}**

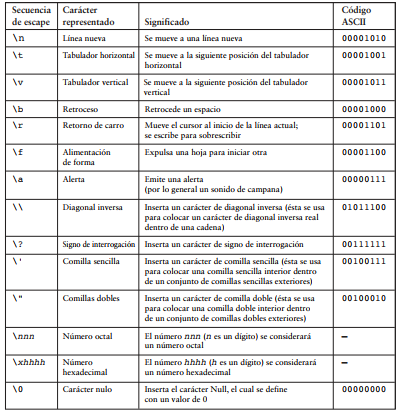
**void mayuscula2(char&c){**

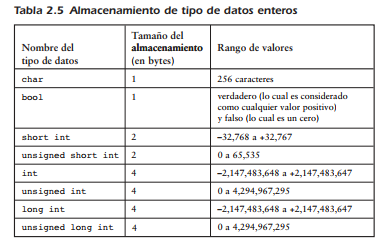
**int aux=int(c);**

**if(aux<97 or aux>122){ return;}**

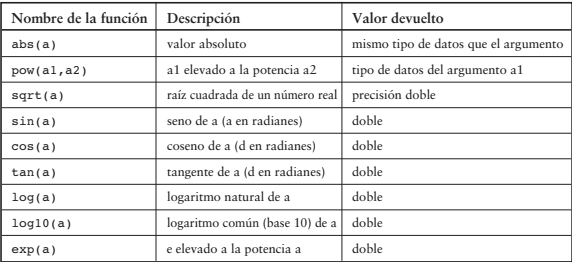
**else{**

**c=char(int(c)-32);}}**





**# include <cmath>**

****

**Calcular mayor y menor conciso con operador ternario**

int mayor(int x,int y){ return (x>=y)? x:y;}

int menor(int x,int y){ return (x<=y)? x:y;}

explicaciones

std::cin es una instancia de std::istream, que representa un flujo de entrada estándar (generalmente el teclado).

Es un **objeto global** declarado en <iostream>, por lo que siempre está disponible para su uso.

Se usa con el operador >> (extracción) para leer datos desde la entrada estándar.

using namespace std; es una directiva en C++ que indica que el programa usará el **espacio de nombres (namespace) std** sin necesidad de escribir std:: antes de cada elemento de la biblioteca estándar.

### **¿Qué es un namespace?**

Un **namespace (espacio de nombres)** en C++ es una forma de agrupar identificadores (funciones, clases, variables, etc.) para evitar conflictos de nombres.

La biblioteca estándar de C++ usa el **namespace std**, donde están definidos elementos como:

std::cout → Para imprimir en pantalla. std::cin → Para leer entrada del usuario.

std::endl → Para un salto de línea. std::vector → Para trabajar con arreglos dinámicos.

std::string → Para manejar cadenas de texto.

### **📌 ¿Por qué son equivalentes?**

int main(int argc, char \*argv[]) int main(int argc, char \*\*argv)

argc: Es un **entero** que indica el número de argumentos pasados por la línea de comandos.

argv: Es un **puntero a un puntero** (char \*\*argv), que apunta a una **lista de cadenas de caracteres**.

🔹 **En C y C++, char \*argv[] y char \*\*argv son la misma cosa.**

* char \*argv[] significa **"un array de punteros a caracteres"**.
* char \*\*argv significa **"un puntero a puntero a caracteres"**.

En realidad, los arreglos ([]) en C/C++ se tratan internamente como punteros en funciones. Por lo tanto:

char \*argv[] ≡ char \*\*argv

Ambas formas indican lo mismo: **una lista de cadenas de caracteres**.

**tipos de argumentos**

**Argumentos posicionales** (Positional Arguments)

**Opciones o flags** (Options/Flags)

**Argumentos con valores** (Options with Values)

**Argumentos combinados** (Combined Arguments)

**Argumentos especiales** (como -- y -)

Operador de comparacion **“==”**

Puede dar **resultado inesperado** debido a **errores de precisión en coma flotante**.  
👉 Se recomienda usar fabs(a - b) < 0.00001 en vez de ==

-- -- -- -- -- -- -- -- -- -- -- -- -- --

Podemos hacer una comparacion de un char con otro, pero no cadenas de char[]

std::string sobrecarga ==, por lo que **se puede comparar directamente**.

⚠️ **No funciona igual con arreglos de char (char[])**:

char palabra[] = "hola";

if (palabra == "hola") { // ❌ Error

std::cout << "No compila\n";}

❌ Esto **NO** compila porque palabra es un puntero (char\*), y == compararía direcciones de memoria, no contenido.

**setfill(j)**: Define el carácter de relleno (j) que se usará para llenar los espacios cuando la salida tenga menos caracteres que el ancho (setw(x)).

**setw(x)**: Establece el ancho mínimo de la próxima salida, pero **NO imprime nada por sí mismo**, solo afecta al siguiente valor que se envíe a cout.

**endl**: Solo imprime un salto de línea (\n), pero no tiene ningún carácter antes, por lo que no hay nada que rellenar con setfill(j).

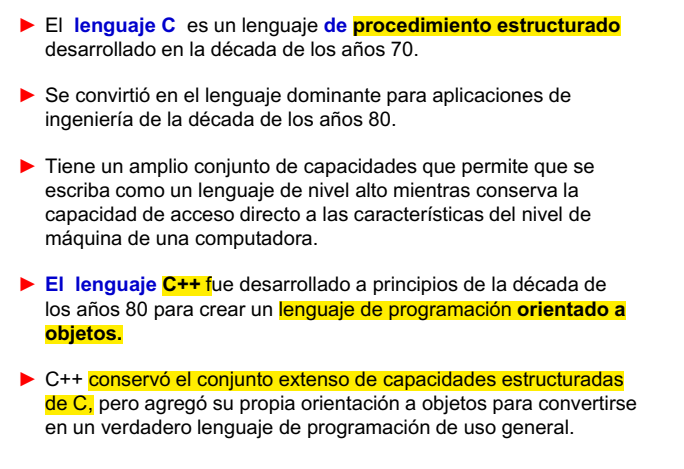
**if ("hola" == "hola")** En muchos compiladores, esto imprimirá **"Son iguales"**, pero **no porque compare el contenido**, sino porque los compiladores suelen optimizar los literales de cadena ("hola") y almacenarlos en la misma dirección de memoria.

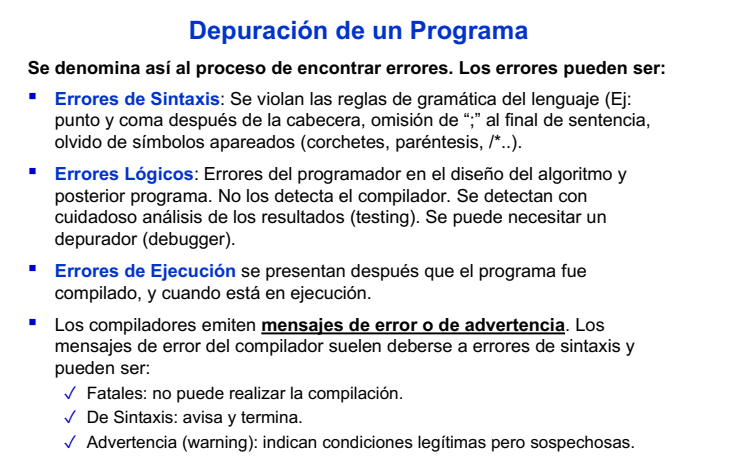
Esto significa que ambas expresiones "hola" apuntan a la misma ubicación en memoria, haciendo que la comparación de punteros con == devuelva true.

presentaciones

El programa fuente se traduce a lenguaje máquina. Esto lo realiza el **compilador** con el sistema operativo

Si no hay errores se obtiene el **programa objeto** (versión en lenguaje de máquina del código fuente).

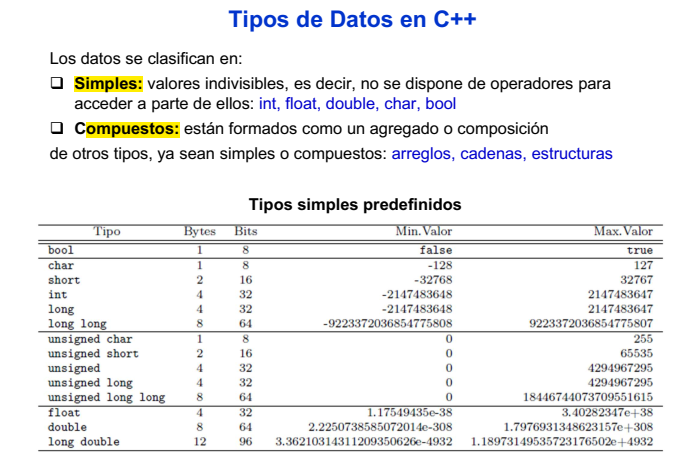
Luego hay que realizar un montaje o enlace (link), que es una combinación del programa objeto con las bibliotecas del compilador. Esto produce un **programa ejecutable** (programa completo en lenguaje de máquina).

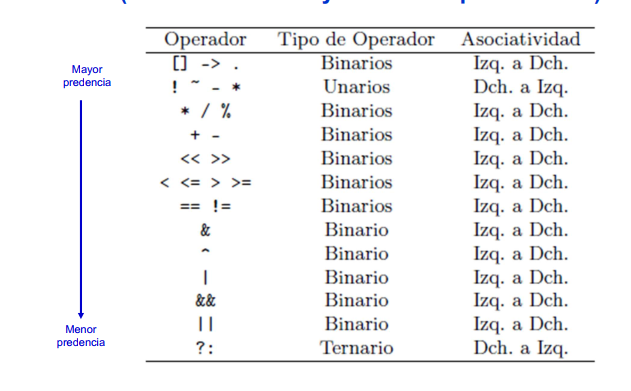


Constantes literales

Constantes definidas **#define PI 3.14**

Constantes declaradas **const float pi=3.14;**





**operador aritmetico - \* / % + -**

-(unitario) de Der a Izq

\* / % de Izq a Der

* - de Izq a Der

Resta operador binario simbolo - frente a una constante unitario

**Operador logico o booleano→**evaluacion de cortocircuito de Izq a Der

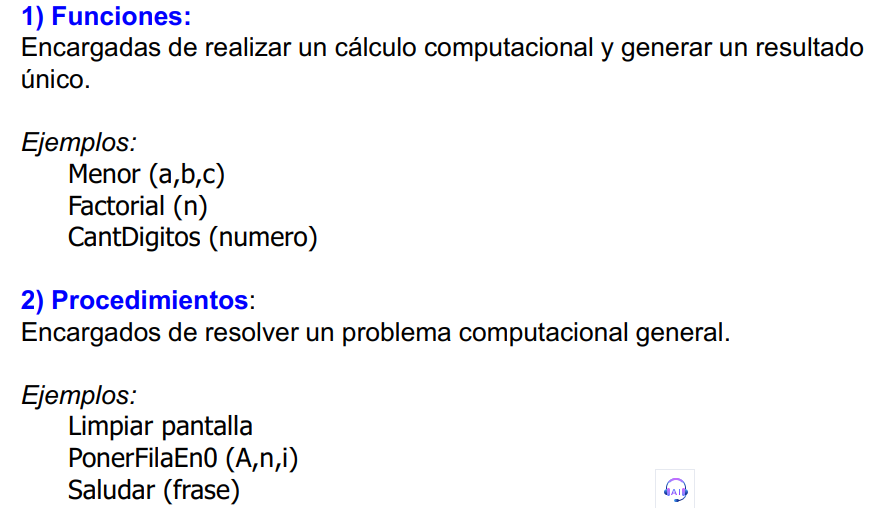
! &&(and) ||(or)

**Operador relacional** < <= > >= == != izq a der

**Operador asignacion** = += -= \*= /= Der a izq

**funciones**

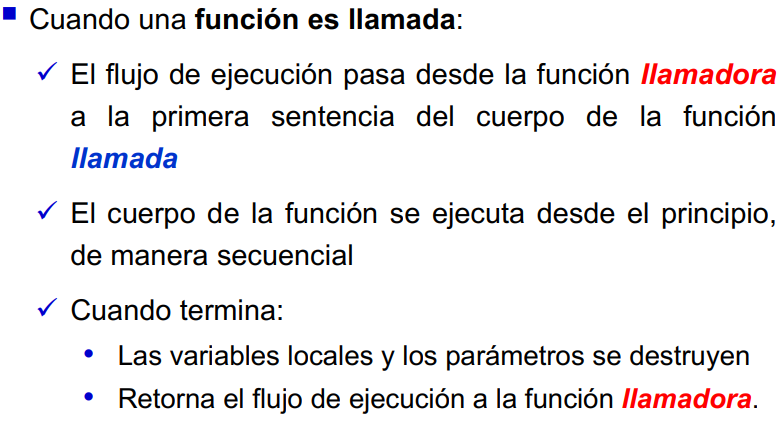
Reuso de código:evita la repetición de código y reduce el tamaño del programa

Mejor uso de la memoria

Se puede **sobrecargar** una funcion si son de distintos tipos de retorno

int sumar(int x);-->**prototipo** para validar funciones sólo se necesita si la definición de la función viene después de su uso en el programa

Funciones sin retorno procedimientos **void**



vectores en adelante

**Void mover(int a[],int tl)**

**MOVER A LA DERECHA**

**int aux=a[tl-1];**

**for(int i=tl-2;i>=0;i--){ a[i+1]=a[i];}**

**a[0]=aux;**

**MOVER A LA IZQUIERDA**

**int aux=a[0];**

**for(int i=1;i<tl;i++){a[i-1]=a[i];}**

**a[tl-1]=aux;**

**Operaciones básicas de vector**

**void insertar(int a[],int& tl,int index,int valor){**

**for(int i=tl-1;i>=index;i--){a[i+1]=a[i];}**

**a[index]=valor;**

**tl++;}**

**-------------------------------------------------------------------------------------**

**bool borrarIndice(int a[],int& tl,int index){**

**if(index>tl-1){ cout<<"posicion inexistente en el arreglo"<<endl;**

**return false;}**

**else{**

**for(int i=index;i<tl-1;i++){a[i]=a[i+1];}**

**tl--;**

**cout<<"borrado existoso"<<endl;**

**return true;}}**

***para borrar por valor podemos organizar un bucle while hasta encontrar el indice y luego otro por separado no interno al while que elimine desde el indice j hasta el tl y lo decremente, para borrar solo un valor***

***La 2da opcion es asi con bucle for interno al while para borrar todos los elementos que existan con ese valor***

**bool borrarValor(int a[],int& tl,int valor){**

**int i=0;**

**bool existe=false;**

**while(i<tl){**

**if(a[i]==valor){existe=true;**

**for(int j=i;j<tl;j++){a[j]=a[j+1]; }**

**tl--;}**

**else{i++;}}**

**if(!existe){cout<<"no existe el valor en el arreglo"<<endl;}**

**return existe; }**

**--------------------------------------------------------------**

**void borrarxRango(int a[],int& tl,int inf,int sup){**

**if(inf>=tl or sup>tl){**

**return;}**

**rango=1+(sup-inf);**

**for(int i=sup+1;i<tl;i++){**

**a[i-rango]=a[i];}**

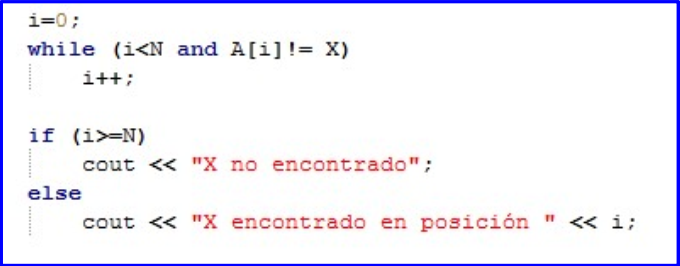
**tl-=rango;**

**}**

**quedan pendiente mejoras de validaciones para que inf y sup sean validos esten dentro del TL e inf no sea mayor que superior**

**Metodos de busqueda**

**Si el arreglo no está ordenado no queda otra que la busqueda secuencial**



**int busquedaSinOrden(int a[],int tl,int elemento){**

**int j=0,existe=-1;**

**while(j<tl and existe==-1){**

**(a[j]==elemento)? existe=j:j++;}**

**return j;**

**}**

**en el mejor caso 0 pasadas en el peor n pasadas**

**----------------------------------------------------------------------------------------------------------------------------------------------------------**

**int busquedaBinaria(int a[],int tl,int elemento){ busqueda con arreglos ordenados**

mejor caso 0 pasadas, peor caso (log2 N)

**int izq=0,der=tl-1;**

**int medio=(tl-1)/2;**

**while(izq<=der and a[medio]!=elemento){**

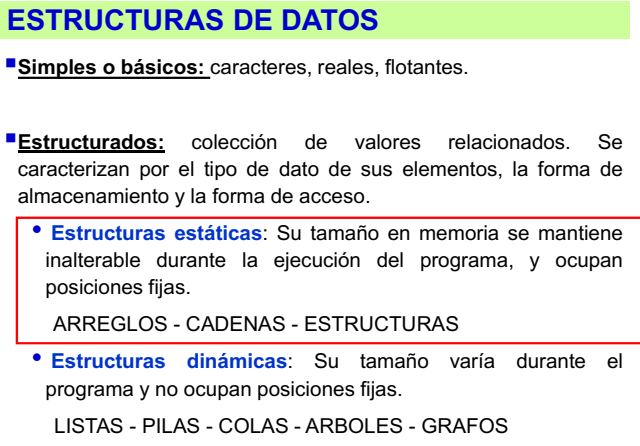
**if(a[medio]<elemento) izq=medio+1;**

**else der=medio-1;**

**medio=(izq+der)/2;}**

**return (der<izq)? -1:medio;}**

presentaciones vectores





* Los datos se llaman elementos (celdas o componentes) del arreglo
* Los elementos del arreglo se almacenan siempre en posiciones consecutivas de la memoria.
* Los elementos se acceden en forma directa, indicando el nombre del arreglo y el subíndice.
* Pueden ser unidimensionales o multidimensionales.
* Trabajan con indexación basada en cero.
* C++ no comprueba que los indices están dentro del tamaño logico definido

**Un arreglo tiene tres partes:**

La dirección (ubicación en memoria de la primera variable indizada)

El tipo base del arreglo (que determina cuánta memoria ocupa cada variable indizada)

El tamaño del arreglo (cantidad de elementos del arreglo)

**Formas de inicialización**

* elemento a elemento int A[6]; A[0]=2; A[1]=6;

si se cargan menos el resto contiene basura como A[2]=1314812384184

* en la declaracion en forma completa int A[6]={1,2,3,4,5,6}
* declararlo int A[]={1,2,3,4};
* inicializar todos en 0 mediante bucle →for(int i=0;i<tl;i++) a[i]=0;
* inicializar cada uno→→for(int i=0;i<tl;i++) cin>>a[i]; en orden o con una variable extra determinar el orden en el que se cargan
* int C[5] = {0}; // Todas las posiciones serán 0
* 💡 **Al asignar {0}, el primer elemento se inicializa en 0 y el resto también, porque los no especificados se llenan con ceros.**

**int c[12]; →&c[0] = 1000 = D → &c[4] = 1000 + 4 \* 4 = 1016**

***La variable A refiere a un espacio de la memoria donde están almacenados en forma contigua los elementos del arreglo.***

***El contenido de A es la dirección donde comienza a almacenarse el arreglo, que también es la dirección del primer elemento del arreglo.***

***&A = &A[0] =D***

***El acceso a cualquier elemento del arreglo tiene la misma complejidad→ACCESO DIRECTO***

typedef float tNuevo[3]

tNuevo a = {12, 34, 45};

tNuevo b = {4, 6, 7};

int a[] es un parámetro de arreglo

Los corchetes sin expresión de índice adentro, son lo que C++ usa 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

para trabajar con una copia explicita se debe hacer localmente en la funcion con un nuevo vector local a la funcion y copiando valor a valor.

para evitar modificarlo pasarlo como parametro de arreglo constante --->int suma(const int a[])

Ordenamiento

**Burbuja mejorado**

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).

**utiliza una marca o señal para indicar que no se ha producido ningún intercambio en una pasada.**

**Seleccion directa**

* 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 (n-1 en total).
* Una desventaja respecto al burbuja con centinela es que **no mejora su rendimiento cuando los datos ya están ordenados o parcialmente ordenados**. En el caso del burbuja con centinela se requiere una única pasada para
* detectar que el vector ya está ordenado y finalizar. En la ordenación por selección se realiza el mismo número de pasadas independientemente de si los datos están ordenados o no.
* Este método es recomendable para un **número pequeño de elementos**.

**Insercion directa**

* En el mejor de los casos, el arreglo estará inicialmente ordenado, entonces el bucle interno (while) sólo ejecuta el paso de comparación, que siempre será falso. Esto implica que en el **mejor de los casos el número de comparaciones sera n-1.**
* En 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 2.**
* A pesar de ser un método ineficiente y recomendable solo para un número pequeño de elementos, es intuitivo y de muy fácil implementación.

**Burbuja mejorado con centinela y - pasadas**

* 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 ahorramos todas las comparaciones de las pasadas restantes.**
* Si el arreglo está inicialmente ordenado, sólo se ejecuta **una pasada sobre el mismo (realizando n-1 comparaciones) y ningún intercambio**.

**MergeSort**

* En éste 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 cada una de las partes, y luego mezclar ambas partes, manteniendo el orden, en un solo vector ordenado
* Tiene la ventaja de tener **complejidad logarítmica: n log (n).**
* Su d**esventaja radica en que se requiere de un espacio extra para el procedimiento**.

**Criterios**

**ESTABILIDAD**: es “estable” si valores iguales en el arreglo guardan su orden relativo.

*Ej: [4a,4b,1,7] resulta [1, 4a,4b,7]*

**NATURALIDAD:** es “natural” *si se tiene en cuenta la posible ordenación del arreglo aunque sea parcial*.

**EFICIENCIA**

**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. Este tipo de métodos sirve para el caso de necesitar ordenar sólo los k ( k<n) primeros elementos.

matrices

**Matrices**

arreglos que permiten varios índices.

A los arreglos bidimensionales se los llama comúnmente matrices, tablas o **arreglos de arreglos**.

A los de 3 dimensiones, se los denomina **cubos** y al resto, en general, se los denomina **multidimensionales**.

**Formato de declaración:** *tipo\_dato nombre-del-arreglo [nroDeFilas] [nroDeColumnas];*

**Inicializacion**

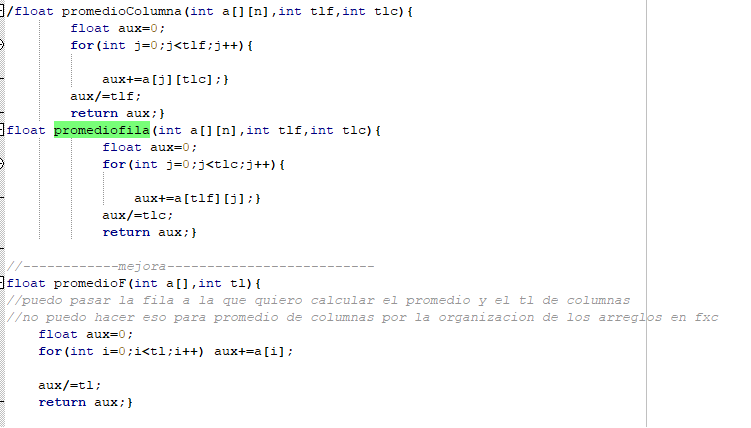
*asignaciones: m1[0] [0] = 0; m1[0] [1] = 0;*

*Enumerando sus elementos en orden de sus dimensiones:*

*tMatriz m1 = { {0,0,0}, {0,0,0} }; cada uno {} es una fila*

*int m1[ ] [3] = { {0,0,0}, {0,0,0} };-->esta inicializacion no es valida requiero conocer ambas componentes [][] uso defines genericos y luego regulo con TL para la fila f y la columna f*

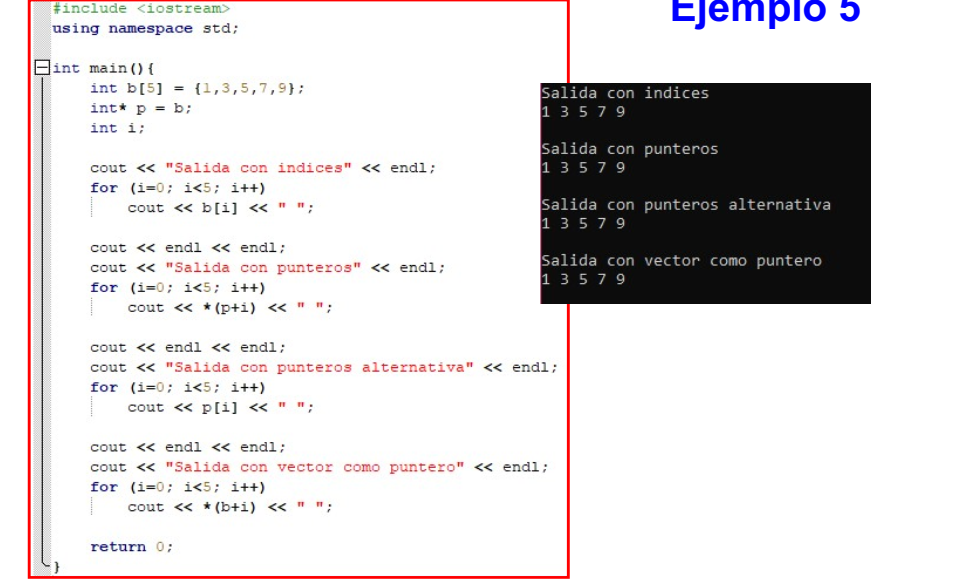
*Los prototipos si pueden ignorar la primera dimension en el argumento (int a[][N],int tlf,int tlc)*

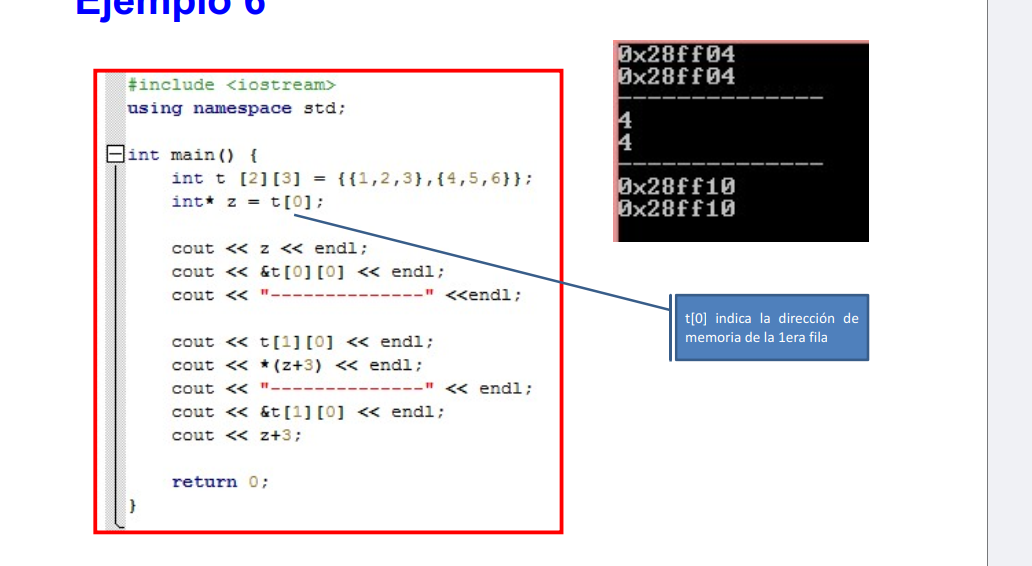


Pasar como argumento una fila completa

quiero pasar la fila 1 puedo hacer → mayorFila(a[1],tlc) o hacer mayorFila(&a[1],tlc) o mayorFila(&a[1][0],tlc)-->siendo tlc el tamaño logico de las columnas

punteros





TDA y STRUCT

Arreglos

Structs

File

String

son estructuras de datos basicas provistas por el lenguaje

Un TDA es un tipo de dato **definido por el programador** que se puede manipular de un modo similar a los tipos de datos provistos por el lenguaje.

Los TDA proporcionan un mecanismo adicional mediante el cual se realiza una **separación clara entre la interfaz y la implementación del tipo de dato.**

**La implementación de un TDA consta de:**

La representación: elección de las estructuras de datos

Las operaciones: elección de los algoritmos

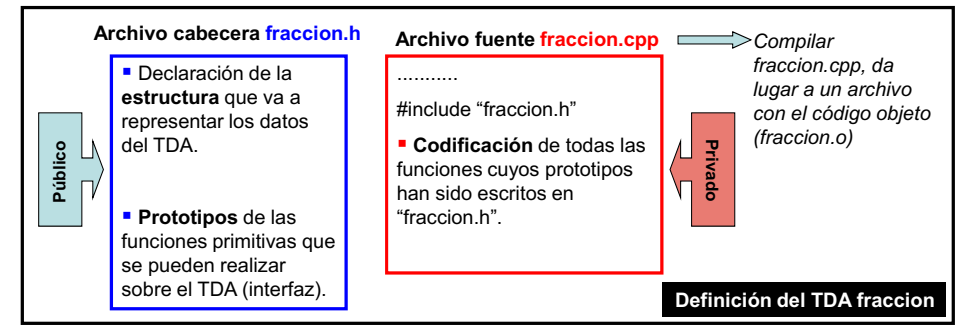
La interfaz del TDA se asocia con las operaciones y datos del TDA, y es visible al exterior.

**operaciones primitivas (interfaz) y - el código (implementación) de dichas operaciones.**

**La implementación queda oculta para el programador “cliente”**

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 (mediante la directiva al preprocesador #include) 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.



Los programas fuentes que vayan a utilizar un TDA necesitan conocer los datos públicos del mismo, es decir su representación y las funciones primitivas que se tienen disponibles, por ello se debe incluir el archivo de cabecera (.h).

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.

Más aún, el archivo fuente (.cpp) que tiene el código de las funciones, puede cambiarse y solo será necesario recompilar, sin que esto afecte al programa que usa el TDA, siempre que la representación e interfaz se mantengan

TDA = Representación (datos) + Operaciones (funciones y procedimientos)

**Ventajas**

**Permite una mejor conceptualización y modelización del mundo real. Clarifica los objetos basados en estructuras y comportamientos comunes.**

**Mejora la robustez yel rendimiento (optimizar tiempos de compilación).**

**Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar la interfaz pública del TDA.**

**Permite la extensibilidad del sistema. Los componentes de software reusables son más fáciles de crear y mantener.**

**STRUCT**

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.

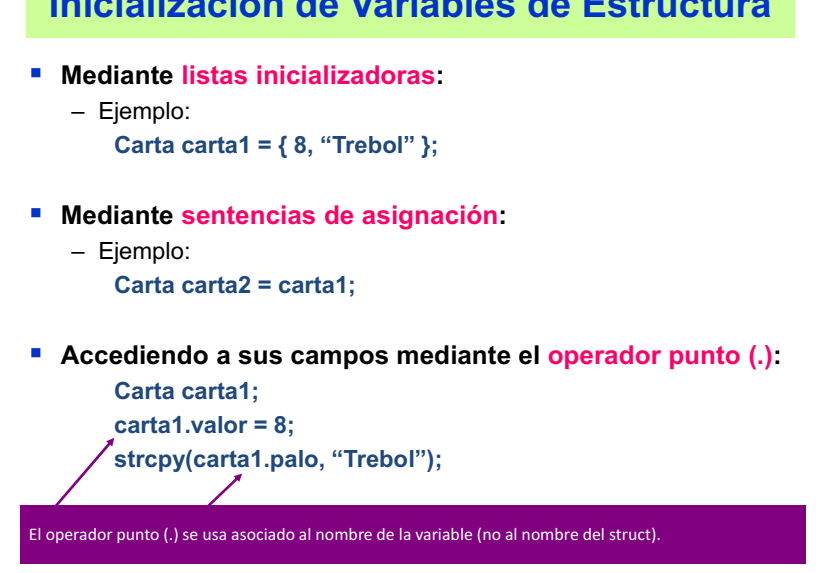
Combinada con punteros, pueden crear listas enlazadas, pilas, colas y árboles.

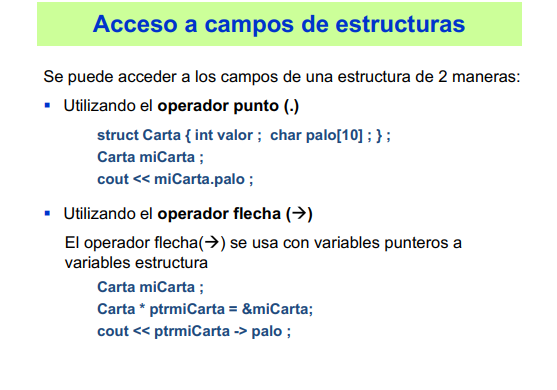
**struct Carta { int valor ;**

**char palo[10] ;} ;**

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 datos, simple o compuesto.



****