



# Introducción a c++ y la programación orientada a objetos

Temperaturas	Códigos	Voltajes
95.75	Z	98
83.0	C	87
97.625	K	92
72.5	L	79
86.25		85
		72

# Arreglos y Vectores

El nombre del arreglo es c

Número de posición del elemento dentro del arreglo c	c[ 0 ]	-45	
	c[ 1 ]	6	
	c[ 2 ]	0	
	c[ 3 ]	72	
Nombre de un elemento individual del arreglo	c[ 4 ]	1543	Valor
	c[ 5 ]	-89	
	c[ 6 ]	0	
	c[ 7 ]	62	
	c[ 8 ]	-3	
	c[ 9 ]	1	
	c[ 10 ]	6453	
	c[ 11 ]	78	

```
int main()
{
    int C[10];
    for (int j =0; j<10;j++){
        C[j]=0;
    }
    cout << "elemento " << " valor" << endl;

    for (int j =0; j<10;j++){
        cout << j << C[j] << endl;
    }
    return 0;
}
```

int A[5]: // A es un arreglo de 5 enteros

**A[j]** j es el numero de la posicion del elemento dentro del arreglo

# Inicialización de un arreglo en una declaración

```
int temp[5] = {98, 87, 92, 79, 85};  
char codigos[6] = {'m', 'u', 'e', 's', 't', 'r', 'a'};  
double pendientes[7] = {11.96, 6.43, 2.58, .86, 5.89, 7.56, 8.22};
```

```
int galones[20] = {19, 16, 14, 19, 20, 18,  
                  12, 10, 22, 15, 18, 17,  
                  16, 14, 23, 19, 15, 18,  
                  21, 5};
```

```
int A[]={1,2,3,4,5};
```

```
int B[5]={1,2,3,4,5};
```

```
int C[7]={1,2,3,4,5,6}; //???
```

```
int main()  
{  
    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };  
  
    cout << "Elemento" << setw( 13 ) << "Valor" << endl;  
  
    for ( int i = 0; i < 10; i++ ){  
        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;  
    }  
    return 0;  
}
```

# Tamaño arreglo con una variable constante y establecimiento de los elementos de un arreglo con cálculos

```
#include<iostream>

using namespace std;

int main()
{
    // la variable constante se puede usar para especificar el tamaño de los arreglos
    const int tamañoArreglo = 10; // debe inicializarse en la declaración

    int s[ tamañoArreglo ]; // el arreglo s tiene 10 elementos

    for ( int i = 0; i < tamañoArreglo; i++){
        s[ i ] = 2 + 2 * i; // establece los valores
    }

    cout << "Elemento" << setw( 13 ) << "Valor" << endl;

    for ( int j = 0; j < tamañoArreglo; j++ )
        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;

    return 0;
}
```

Esto compila?

E. Código  
example1\_barras

E. Código  
example2\_contadores (datos)

E. Código  
example3\_encuesta

# Uso de arreglos tipo carácter para almacenar y manipular cadenas

```
char cadena1[] = "first";  
char cadena1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

## caracter nulo '\0'

Todas las cadenas representadas mediante arreglos de caracteres terminan con este carácter.

Sin él, este arreglo representaría tan sólo un arreglo de caracteres, no una cadena

```
char cadena2[ 20 ];  
cin >> cadena2;
```

Es responsabilidad del programador asegurar que el arreglo en el que se coloque la cadena sea capaz de contener cualquier cadena que el usuario escriba en el teclado.

```
int main(){  
    char cadena1[ 20 ];  
    char cadena2[] = "literal de cadena";  
  
    // lee la cadena del usuario y la coloca en el arreglo cadena1  
    cout << "Escriba la cadena \"hola todos\": ";  
    cin >> cadena1;  
  
    cout << "cadena1 es: " << cadena1 << "\ncadena2 es: " << cadena2;  
    cout << "\ncadena1 con espacios entre caracteres es:\n";  
  
    // imprime caracteres hasta llegar al caracter nulo  
    for ( int i = 0; cadena1[ i ] != '\0'; i++){  
        cout << cadena1[ i ] << ' ';  
    }  
  
    cin >> cadena1; // lee "todos"  
    cout << "\ncadena1 es: " << cadena1 << endl; // NOTE: cuidado con  
    la linea fantasma. aca debe usar getline(cin,string)  
  
    return 0;  
}
```

## Arreglos locales estáticos

```
void inicArregloStatic( void )      E. Codigo
{                                  example5_arreglosstati
// inicializa con 0 la primera vez que se llama a la función
    static int arreglo1[ 3 ];

    cout << "\nValores al entrar en inicArregloStatic:\n";

    for ( int i = 0; i < 3; i++ ){
        cout << "arreglo1[" << i << "] = " << arreglo1[ i ] << "
";
    }

    cout << "\nValores al salir de inicArregloStatic:\n";

    // modifica e imprime el contenido de arreglo1
    for ( int j = 0; j < 3; j++ )
        cout << "arreglo1[" << j << "] = " << ( arreglo1[ j ] +=
5 ) << " ";
    }
```

Podemos aplicar `static` a la declaración de un arreglo local, de manera que el arreglo no se cree e inicialice cada vez que el programa llame a la función, y no se destruya cada vez que termine la función en el programa. Esto puede mejorar el rendimiento, en especial cuando se utilizan arreglos extensos.

## Paso de arreglos a funciones

**C++ pasa los arreglos a las funciones por referencia.**

las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales.

El paso de arreglos por referencia tiene sentido por cuestiones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Para los arreglos extensos que se pasan con frecuencia, esto requeriría mucho tiempo y una cantidad considerable de almacenamiento para las copias de los elementos del arreglo.

Observe la extraña apariencia del prototipo  
**void FunA( int [], int );** //Arreglo y tamaño

**C++ ignoran los nombres de las variables en los prototipos**

**void FunArreglo( int NameA[], int VariableSizeA);**

E. Codigo example6\_modificarA

```

#include <iostream>
#include <iomanip>

using namespace std;

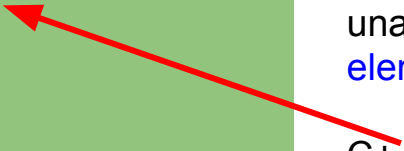
void tratarDeModificarArreglo( const int [] );

int main()
{
    int a[] = { 10, 20, 30 };

    tratarDeModificarArreglo( a );
    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
    return 0;
}

void tratarDeModificarArreglo( const int b[] )
{
    b[ 0 ] /= 2; // error de compilación
    b[ 1 ] /= 2;
    b[ 2 ] /= 2;
}

```



## Principio de menor privilegio.

Las funciones no deben recibir la capacidad de modificar un arreglo, a menos que sea absolutamente necesario

se puede encontrar con situaciones en las que una función **no tenga permitido modificar los elementos de un arreglo**.

C++ cuenta con el calificador de tipos **const**.

Cuando una función especifica un parámetro tipo arreglo al que se antepone el calificador **const**, los elementos del arreglo se hacen constantes en el cuerpo de la función

E. Código librocalicar1 (clase6)

E. Código  
example7\_busquedalineal

E. Código  
example8\_busquedainsercion

# Arreglos multidimensionales

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Fila 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Fila 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

Diagram illustrating the structure of a 2D array. The array is represented as a table with rows (Fila 0, Fila 1, Fila 2) and columns (Columna 0, Columna 1, Columna 2, Columna 3). Each element is accessed using the format `a[ row ][ column ]`. Arrows point from the labels 'Subíndice de columna', 'Subíndice de fila', and 'Nombre del arreglo' to the corresponding parts of the array notation in the table.

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

`b[ 0 ][ 0 ] = 1`, `b[ 0 ][ 1 ] = 2`, `b[ 1 ][ 0 ] = 3`, `b[ 1 ][ 1 ] = 4`,

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

`b[ 0 ][ 0 ] = 1`, `b[ 0 ][ 1 ] = 0`, `b[ 1 ][ 0 ] = 3` y `b[ 1 ][ 1 ] = 4`

E. Código librocalicar2 (clase6)

```
void imprimirArreglo( const int a[ 3 ] );
```

```
int main()  
{
```

```
int arreglo1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
int arreglo2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
```

```
int arreglo3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
```

```
cout << "Los valores en arreglo1 por fila:" << endl;
```

```
imprimirArreglo( arreglo1 );
```

```
cout << "\nLos valores en arreglo2 por fila:" << endl;
```

```
imprimirArreglo( arreglo2 );
```

```
return 0;
```

```
}
```

```
void imprimirArreglo( const int a[ 3 ] )
```

```
{
```

```
// itera a través de las filas del arreglo
```

```
for ( int i = 0; i < 2; i++ ){
```

```
// itera a través de las columnas de la fila actual
```

```
for ( int j = 0; j < 3; j++ )
```

```
    cout << a[ i ][ j ] << ' ';
```

```
    cout << endl; // empieza nueva línea de salida
```

```
}
```

```
}
```



# La clase vector de STL (standard template library)

Una de las dificultades del lenguaje C es la implementación de contenedores (vectores, listas enlazadas, conjuntos ordenados) genéricos, de fácil uso y eficaces. Para que estos sean genéricos por lo general estamos obligados a recurrir a punteros genéricos (void \*) y a operadores de cast. Es más, cuando estos contenedores están superpuestos unos a otros (por ejemplo un conjunto de vectores) el código se hace difícil de utilizar.

Para responder a esta necesidad, la STL (standard template library) **implementa un gran número de clases template describiendo contenedores genéricos para el lenguaje C++**.

**std::pair<T1,T2>**  
**std::list<T,...>**  
**std::vector<T,...>**  
**std::set<T,...>**  
**std::map<K,T,...>**

```
#include <iostream>
#include <string>
#include <list>
```

```
int main(){
```

```
    std::list<int> ma_lista;
    ma_lista.push_back(4);
    ma_lista.push_back(5);
    ma_lista.push_back(4);
    ma_lista.push_back(1);
```

```
    std::list<int>::const_iterator lit (mi_lista.begin()),
    lend(mi_lista.end());
```

```
    for(;lit!=lend;++lit) {
        std::cout << *lit << ' ';
    }
```

```
    std::cout << std::endl;
    return 0;
```

```
}
```

# La clase vector de STL (standard template library)

Funciones (métodos de clase) y operaciones	Descripción
<code>vector&lt;TipoDatos&gt; nombre</code>	Crea un vector vacío con tamaño inicial dependiente del compilador
<code>vector&lt;TipoDatos&gt; nombre(fuente)</code>	Crea una copia del vector fuente
<code>vector&lt;TipoDatos&gt; nombre(n)</code>	Crea un vector de tamaño <i>n</i>
<code>vector&lt;TipoDatos&gt; nombre(n, elem)</code>	Crea un vector de tamaño <i>n</i> con cada elemento inicializado como <i>elem</i>
<code>vector&lt;TipoDatos&gt; nombre(src.beg, src.end)</code>	Crea un vector inicializado con elementos de un contenedor fuente que comienza en <i>src.beg</i> y termina en <i>src.end</i>
<code>~vector&lt;TipoDatos&gt;()</code>	Destruye el vector y todos los elementos que contiene
<code>nombre[índice]</code>	Devuelve el elemento en el índice designado, sin comprobación de límites
<code>nombre.at(índice)</code>	Devuelve el elemento en el argumento del índice especificado, sin comprobación de límites en el valor del índice
<code>nombre.front()</code>	Devuelve el primer elemento en el vector
<code>nombre.back()</code>	Devuelve el último elemento en el vector
<code>dest = src</code>	Asigna todos los elementos del vector <i>src</i> al vector <i>dest</i>

E. Código example9\_vector

# Generalidades acerca del manejo de excepciones

Realizar una tarea

Si la tarea anterior no se ejecutó correctamente

Realizar el procesamiento de los errores

Realizar la siguiente tarea

Si la tarea anterior no se ejecutó correctamente

Realizar el procesamiento de los errores

...

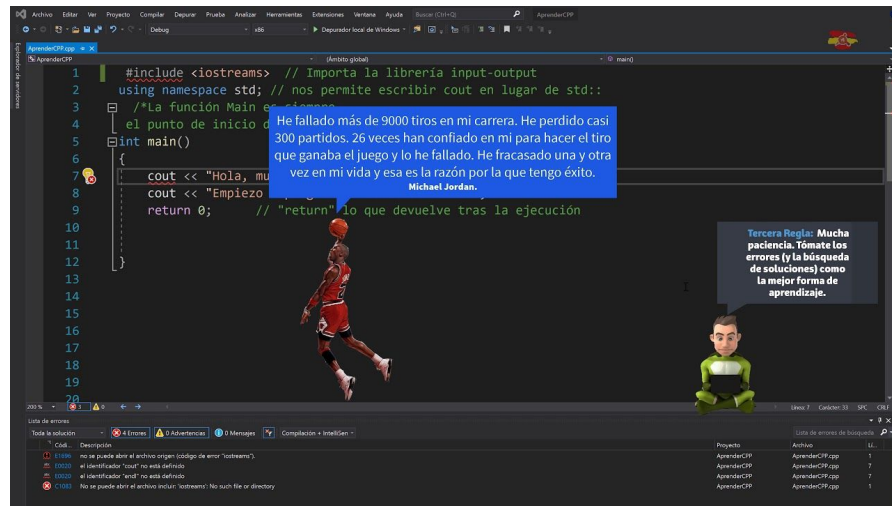
...

...

...

Aunque esta forma de manejo de excepciones funciona

Si los problemas potenciales ocurren con poca frecuencia, **al entremezclar la lógica del programa y la lógica del manejo de errores se puede degradar el rendimiento del programa**, ya que éste debe realizar pruebas (tal vez con frecuencia) para determinar si la tarea se ejecutó en forma correcta, y si se puede llevar a cabo la siguiente tarea.



El manejo de excepciones proporciona un mecanismo estándar para procesar los errores. **Esto es especialmente importante cuando se trabaja en un proyecto con un equipo extenso de programadores.**

```

int main()
{
    int x = 1;
    // instrucciones preliminares.....

    cout << "antes del try \n";
    try {
        cout << "dentro del try \n";
        if (x < 0){
            throw x; // el tipo de excepcion sera un entero
            cout << "despues del throw (NO se debe
ejecutar) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "despues del catch (esto seguira
ejecutandose) \n";

    return 0;
}

```

E. Codigo example\_throw.cpp (clase6)  
E. Codigo example2.cpp (clase6)

**bloque try:** Encierra instrucciones que podrian ocasionar excepciones e instrucciones que se debria omitir si ocurren excepciones

**Palabra clave throw:** representa el tipo de excepcion que se va a lanzar

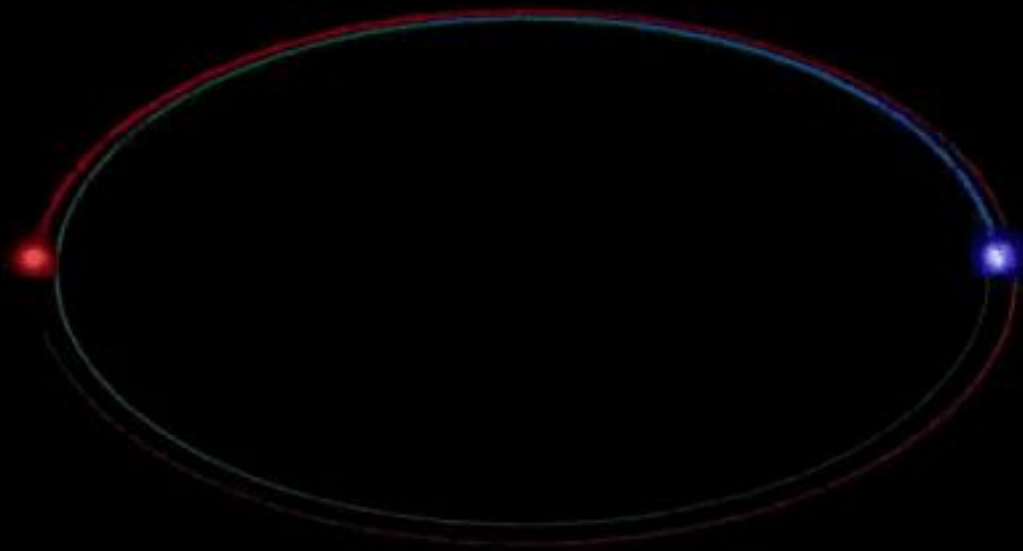
**catch:** sirven como manejadores de excepciones para cualesquiera excepciones lanzadas por las instrucciones en el bloque try

```

cout << endl;
cout << "aca una excepcion general. \n";

try {
    throw 10;
}
catch (char excp) {
    cout << "Caught " << excp;
}
catch (...) {
    cout << "por default \n";
}

```





**Detector Model** ?

- Tracker Barrels ☐
- Tracker Endcaps ☐
- ECAL Barrel ☒
- ECAL Endcaps ☐
- ECAL Preshower ☐
- HCAL Barrel ☐
- HCAL Endcaps ☐
- HCAL Outer ☒
- HCAL Forward ☐
- Drift Tubes (muon) ☐
- Cathode Strip Chambers (muon) ☐
- Resistive Plate Chambers (muon) ☐

**Tracking** ?

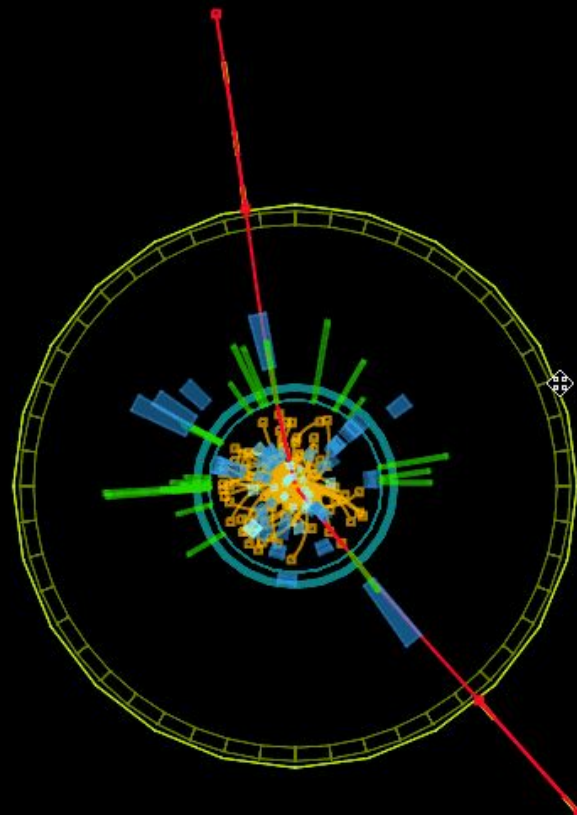
- Tracks (reco.) ☒
- Clusters (Si Pixels) ☐
- Clusters (Si Strips) ☐
- Rec. Hits (Tracking) ☐

**ECAL** ?

- Barrel Rec. Hits ☒ ▶
- Endcap Rec. Hits ☐ ▶
- Preshower Rec. Hits ☐ ▶

**HCAL** ?

- Barrel Rec. Hits ☒ ▶
- Endcap Rec. Hits ☒ ▶
- Forward Rec. Hits ☒ ▶
- Outer Rec. Hits ☐ ▶



# La clase string: flujos de cadena

```
string texto( "Hola" );  
string nombre( 8, 'x' ); // cadena de 8 caracteres 'x'  
string mes = "Marzo"; // igual que: string mes( "Marzo" );
```

**string no proporciona conversiones de int o char a string en una definición string.**

```
string error1 = 'c';  
string error2( 'u' );  
string error3 = 22;  
string error4( 8 );
```

```
string objetoString;  
cin >> objetoString;
```

**La función getline también se sobrecarga para objetos string:**

```
getline( cin, cadena1 );
```

## String in C++

C style String

C++ style String

```
Char e[] = "geeks"  
Char e1[] = {'g', 'f', 'g', '10'};  
Char * C = "geeksforgeeks";
```

```
String str = ("gfg");  
String str = "" g;  
String str ; str = "gfg";
```



- E. Código example3\_concatenacion.cpp (clase6)
- E. Código example4\_comparar.cpp (clase6)
- E. Código example5\_subcadenas.cpp (clase6)
- E. Código example6\_caracteristicas.cpp (clase6)