



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 número de la posición del elemento dentro del arreglo

Arreglos y Arreglos stl

```
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;
}
```

Codigo: example0.cpp

```
#include <array>
```

```
int main() {
    array<int, 10> n; //n es un arreglo de 10 enteros

    // initialize elements of array n to 0
    for (size_t i{0}; i < n.size(); ++i) {
        n[i] = 0; //establece el elemento en la ubicación i a 0
    }

    cout << "Elemento" << setw(10) << "Valor" << endl;
    // imprime el valor de cada elemento del arreglo
    for (size_t j{0}; j < n.size(); ++j) {
        cout << setw(7) << j << setw(10) << n[j] << endl;
    }
    return 0;
}
```

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?

Si no se asigna un valor a una variable constante cuando se declara es un error de compilación.

E. Código
example1_barras

E. Código
example2_contadores (datos)

E. Código
example3_encuesta

"C++ no cuenta con comprobación de límites para evitar que la computadora haga referencia a un elemento que no existe."

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_arreglosstatic
// 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

instrucción for basada en rango

```
int main() {  
    array<int, 5> items{1, 2, 3, 4, 5};  
  
    cout << "items before modification: ";  
    for (int item : items) {  
        cout << item << " ";  
    }  
  
    for (int& itemRef : items) {  
        itemRef *= 2;  
    }  
  
    cout << "\nitems after modification: ";  
    for (int item : items) {  
        cout << item << " ";  
    }  
  
    cout << endl;  
}
```



```

#include <iostream>
#include <iomanip>

using namespace std;

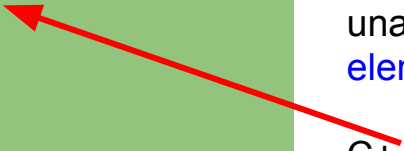
void tratarDeModificarArreglo( const int [] );

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](#)

E. Código [example8_sort_new](#)

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`

Tarea, no calificable.

¿Como se haria esto con el template "array"?

```
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++ )
```

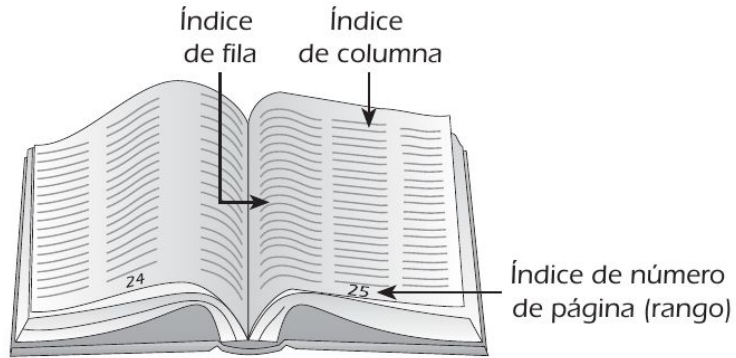
```
    cout << a[ i ][ j ] << ' ';
```

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

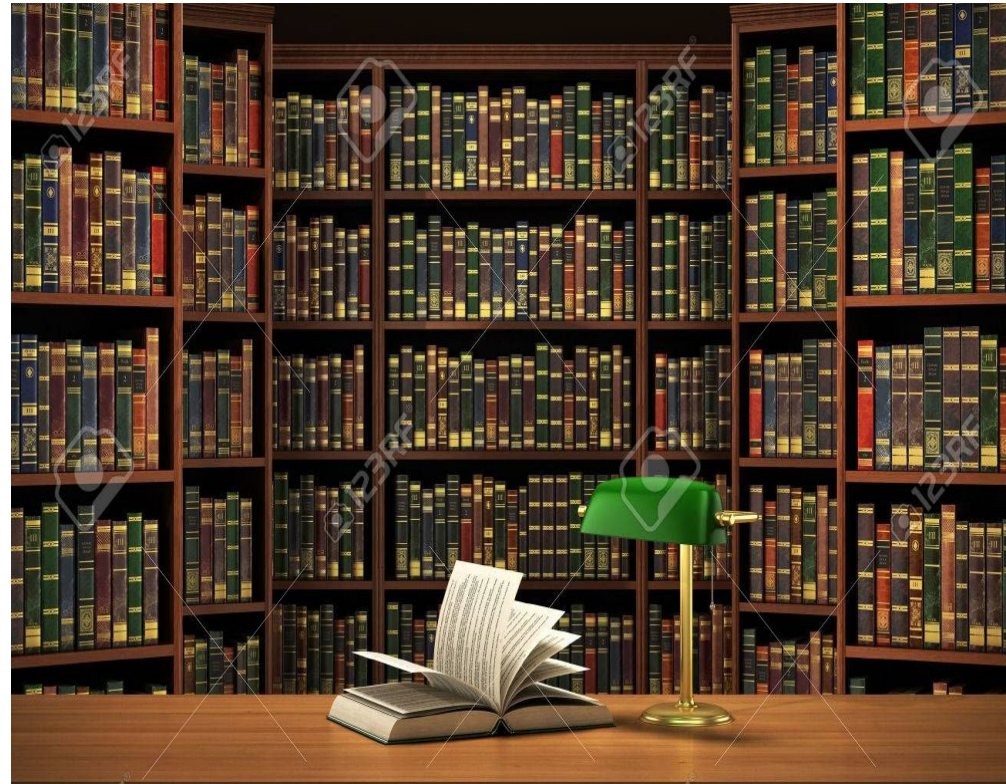
```
}
```

```
}
```

Arreglos multidimensionales



© Can Stock Photo - csp33529995



E. Código librocalicar2 (clase6)

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;
}
```

codigo: myList

La clase vector de STL (standard template library)

Funciones (métodos de clase) y operaciones	Descripción
<code>vector<TipoDatos> nombre</code>	Crea un vector vacío con tamaño inicial dependiente del compilador
<code>vector<TipoDatos> nombre(fuente)</code>	Crea una copia del vector fuente
<code>vector<TipoDatos> nombre(n)</code>	Crea un vector de tamaño <i>n</i>
<code>vector<TipoDatos> nombre(n, elem)</code>	Crea un vector de tamaño <i>n</i> con cada elemento inicializado como <i>elem</i>
<code>vector<TipoDatos> 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<TipoDatos>()</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

E. Código example10_fibo

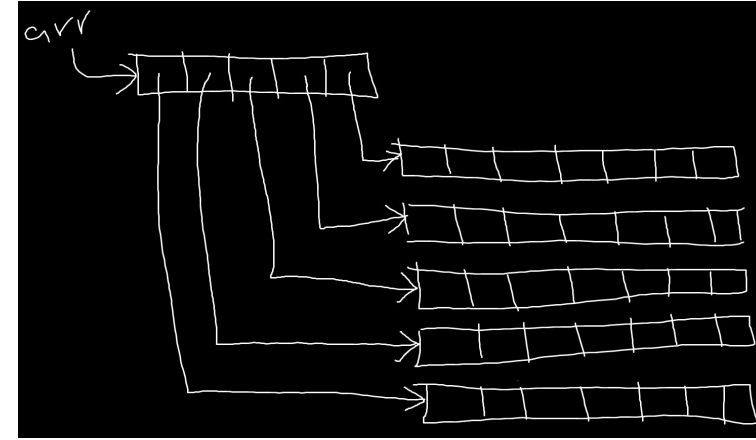
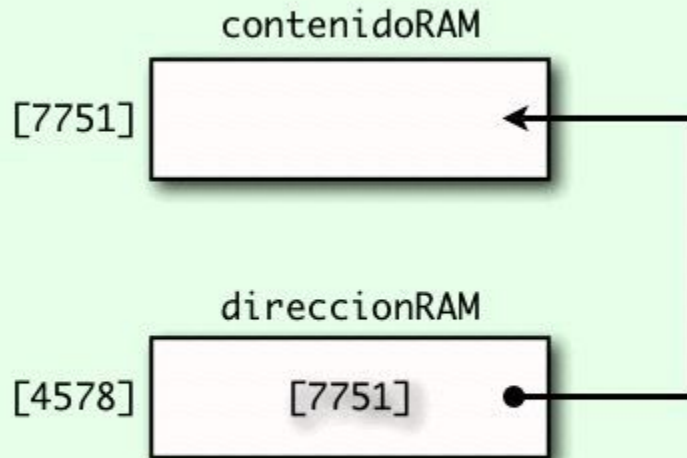
Podemos poner un poco de luz para ver la grandeza de Khazad-dûm



- ¿Qué es este nuevo horror Gandalf?
- Apuntadores, un demonio del mundo antiguo.
¿es éste un poder que alguno de ustedes puede enfrentar?



Apuntadores, los arreglos y las cadenas estilo C

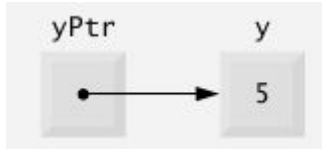


Los apuntadores nos dan acceso al funcionamiento interno de la computadora y la estructura de almacenamiento básico

Declaraciones e inicialización de variables apuntdores

```
Int y = 5; // Decalara la variable
int *yPtr; //Decalara la variable apuntdor

yPtr = &y; // asigna la direcion de y a yPtr
```



variables que se usan para almacenar direcciones de memoria

```
double *B_mass, *B_px, *B_py, *B_pz;

int*    J_mass, J_px, Jpy, Jpz; ???
```

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

```
int main()
{
    int a;
    int *aPtr; // aPtr es un int * -- apuntdor a un entero

    a = 7;
    aPtr = &a; // aca asignamos la direcci3n de "a" a "aPtr"

    cout << "La direccion de a es " << &a
          << "\nEl valor de aPtr es " << aPtr;
    cout << "\n\nEl valor de a es " << a
          << "\nEl valor de *aPtr es " << *aPtr;
    cout << "\n\nDemostracion de que * y & son inversos "
          << "uno del otro.\n&*aPtr = " << &*aPtr
          << "\n*&aPtr = " << *&aPtr << endl;

    return 0;
}
```


Declaraciones e inicialización de variables apunadores

Inicialice todos los punteros para evitar que apunten a áreas de memoria desconocidas o no inicializadas.

Pointers should be initialized to **nullptr** (added in C++11) or to a memory address either when they're declared or in an assignment.

En versiones anteriores de C++, el valor especificado para un **puntero nulo** era **0** o **NULL**.

```
int y{5}; // declara la variable y
int* yPtr{nullptr}; // declara la variable puntero yPtr
yPtr = &y; // asigna la dirección de y a yPtr
```

```
*yPtr = 9;
cin >> *yPtr;
```

Seguimiento2, tarea1

4 - 3 - 5 - 2 - 1

Tenemos 5 elementos. Es decir, TAMAÑO toma el valor 5. Comenzamos comparando el primero con el segundo elemento. 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

3 - 4 - 5 - 2 - 1

Ahora comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Continuamos con el tercero y el cuarto: 5 es mayor que 2. Intercambiamos y obtenemos:

3 - 4 - 2 - 5 - 1

Comparamos el cuarto y el quinto: 5 es mayor que 1. Intercambiamos nuevamente:

3 - 4 - 2 - 1 - 5

Repitiendo este proceso vamos obteniendo los siguientes resultados:

3 - 2 - 1 - 4 - 5

2 - 1 - 3 - 4 - 5

1 - 2 - 3 - 4 - 5





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 ?

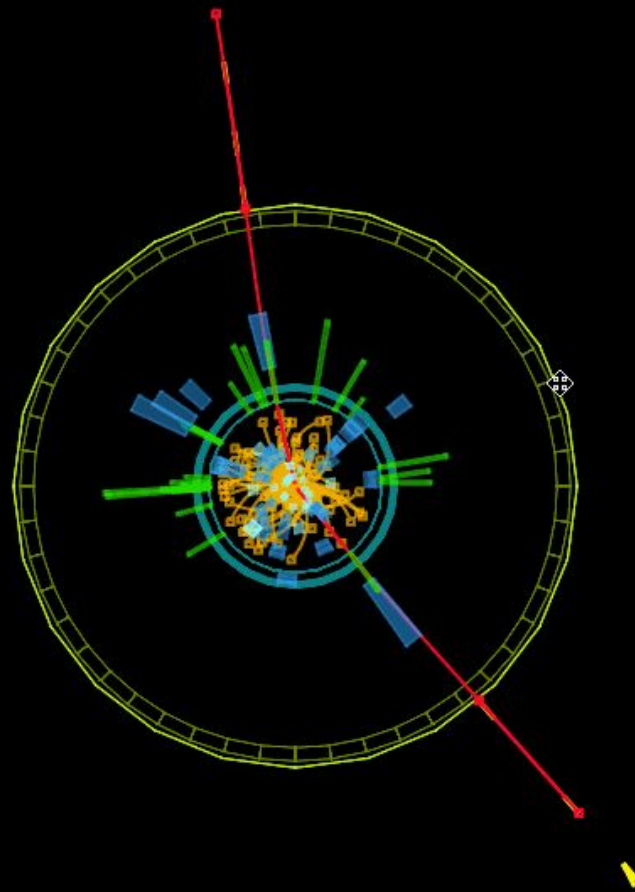
- Tracks (reco.) ☒
- Clusters (Si Pixels) ☐
- Clusters (Si Strips) ☐
- Rec. Hits (Tracking) ☐

ECAL ?

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

HCAL ?

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



Paso de argumentos a funciones por referencia mediante apuntadores

Paso por referencia

Type Fun (type &, type &) ;

mediante apuntadores

Type Fun (type *, type *) ;

En general, para funciones "sencillas", gana la conveniencia de la notación y se usan referencias. Sin embargo, **al transmitir arreglos** a funciones el compilador transmite de manera automática una dirección. **Esto dicta que se usarán variables apunadoras para almacenar la dirección.**

E. Codigo example2_ArgPorReren.cpp (clase7)

Uso de const con apuntadores

-Si un valor no cambia (o no debe cambiar) en el cuerpo de una función que lo recibe, el parámetro se debe declarar const para asegurar que no se modifique por accidente.

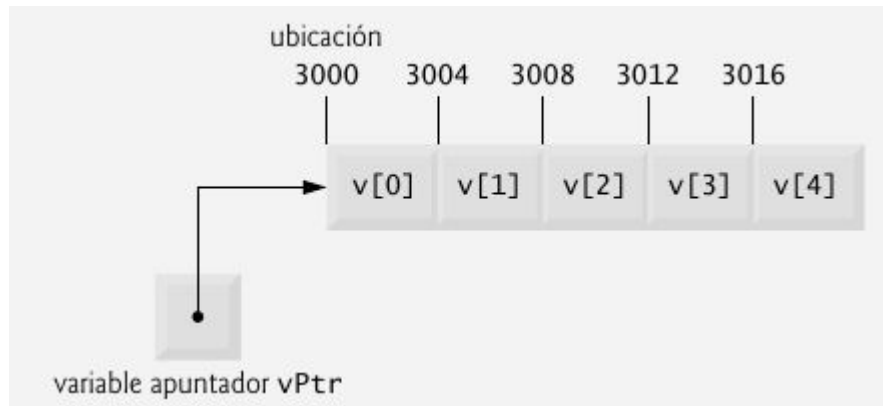
-Antes de usar una función, compruebe su prototipo para determinar los parámetros que puede modificar.

- Cuando el compilador encuentra el parámetro de una función para un arreglo unidimensional de la **forma int b[] , convierte el parámetro a la notación de apuntador int *b.** Ambas formas de declarar un parámetro de función como arreglo unidimensional son intercambiables.

E. Codigo example3_Const.cpp (clase7)

E. example4_OrdeSelec.cpp (clase7)

E. example5_sizeof.cpp (clase7)



```
int *vPtr = v;  
int *vPtr = &v[ 0 ];
```

```
vPtr +=2 ;    (3000 + 2*4 = 3008 )
```

En el arreglo v, vPtr apuntaría ahora a v[2]

new example:

```
vPtr +=4 ;
```

En el arreglo v, vPtr apuntaría ahora a v[4]

```
vPtr -=3 ;   ???
```

Aritmética de apuntadores

Se pueden realizar varias operaciones aritméticas con los apuntadores. Un apuntador se puede incrementar (++) o decrementar (--), se puede sumar un entero a un apuntador (+ o +=), se puede restar un entero de un apuntador (- o -=), o se puede restar un apuntador de otro apuntador del mismo tipo.

NOTA: La mayoría de las computadoras de la actualidad tienen enteros de dos o de cuatro bytes. Algunos de los equipos más recientes utilizan enteros de ocho bytes. Debido a que los resultados de la aritmética de apuntadores dependen del tamaño de los objetos a los que apunta un apuntador, **la aritmética de apuntadores es dependiente del equipo.**

Relación entre apuntadores y arreglos

Los arreglos y los apuntadores están estrechamente relacionados en C++ y se pueden utilizar de manera casi intercambiable. **El nombre de un arreglo se puede considerar como un apuntador constante.** Los apuntadores se pueden utilizar para realizar cualquier operación en la que se involucren los subíndices de arreglos.

```
int b[ 5 ];  
int *bPtr;
```

E. Código example6_notacionApun.cpp (clase7)

```
bPtr = b; // asigna la dirección del arreglo b a bPtr  
bPtr = &b[ 0 ]; // también asigna la dirección del arreglo b a bPtr
```

El elemento `b[3]` del arreglo se puede referenciar de manera alternativa con la siguiente expresión de apuntador:

`*(bPtr + 3)` y en general

`b[i] = *(bPtr + i)`

E. Códigos
example7_copystringArreglos.cpp
example9_ApuntFunc.cpp
example10_ArregApuntFun.cpp
(clase7)



NOTA1: El nombre del arreglo (**que es const de manera implícita**) se puede tratar como apuntador y se puede utilizar en la aritmética de apuntadores. Por ejemplo, la expresión `*(b + 3)`

Sin embargo

`b += 3`

produce un error de compilación, trata de modificar una constante.

NOTA2: Los apuntadores pueden usar subíndices de la misma forma que los arreglos. Por ejemplo, la expresión `bPtr[1]`