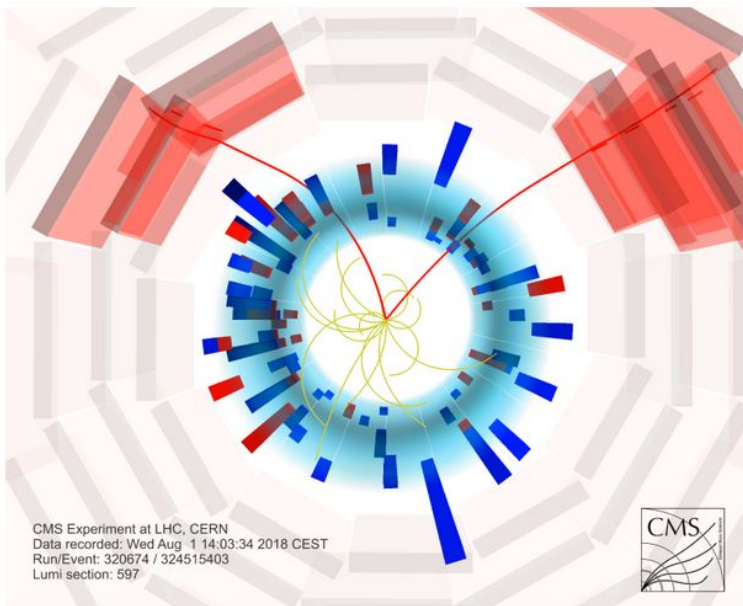


ROOT

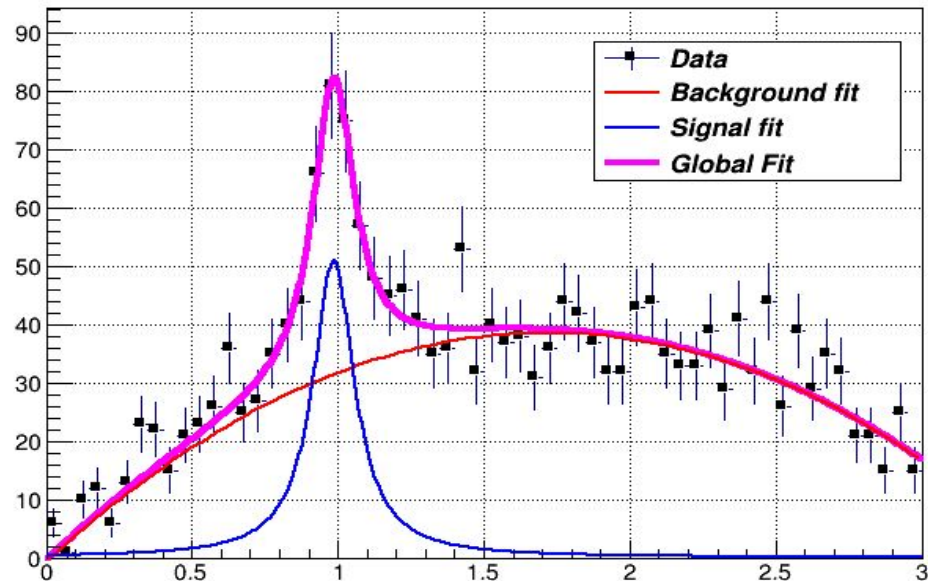
An Object-Oriented
Data Analysis Framework



CMS Experiment at LHC, CERN
Data recorded: Wed Aug 1 14:03:34 2018 CEST
Run/Event: 320674 / 324515403
Lumi section: 597



Lorentzian Peak on Quadratic Background



Big data With ROOT CERN

Jhovanny Andres Mejia Guisao

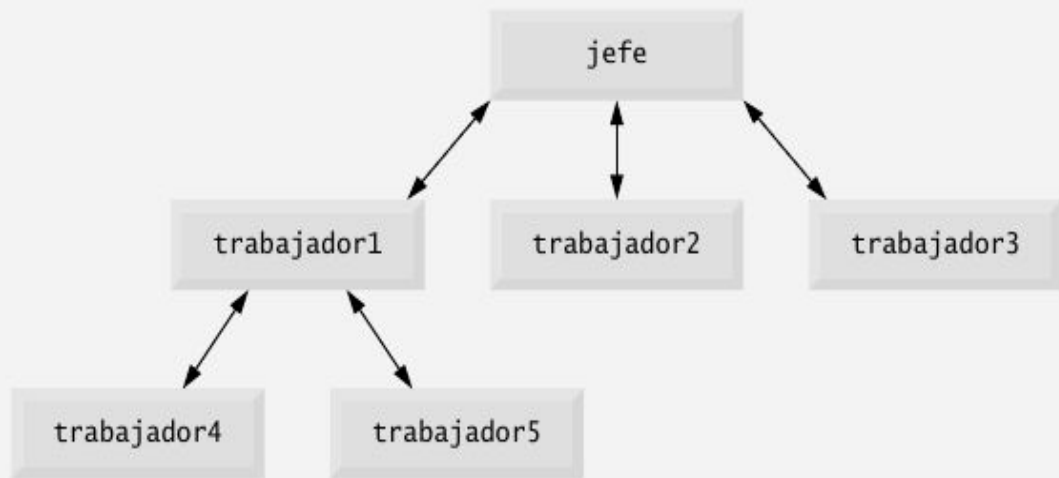
Centro Interdisciplinario de Investigación
y Enseñanza de la Ciencia

What we hope to discuss about scientific data analysis?

- **Advanced graphical user interface**
- **Interpreter for the C++ programming language**
- **Persistency mechanism for C++ objects**
- **Used to write every year petabytes of data recorded by the Large Hadron Collider experiments**

Input and plotting of data from measurements and fitting of analytical functions.

Continuamos con las funciones



divide y vencerás

Para promover la reutilización de software, toda función debe limitarse a realizar una sola tarea bien definida, y el nombre de la función debe expresar esa tarea con efectividad. Dichas funciones facilitan la escritura, prueba, depuración y mantenimiento de los programas.

Una pequeña función que realiza una tarea es más fácil de probar y depurar que una función más grande que realiza muchas tareas.

Si no puede elegir un nombre conciso que exprese la tarea de una función, tal vez ésta esté tratando de realizar demasiadas tareas diversas. Por lo general, es mejor descomponer dicha función en varias funciones más pequeñas.

Continuamos con las funciones

```
#include<iostream>

using namespace std;

//int VolumenCaja(int , int , int );
int VolumenCaja(int = 1, int = 1, int =1 );

int main()
{

    cout << "el volumen predeterminado es " << VolumenCaja()<<
endl;
    cout << endl;

    cout << "el volumen que le damos es " << VolumenCaja(2,2,2)<<
endl;

    return 0;
}

int VolumenCaja(int L, int A , int P ){

    return L*A*P;

}
```

Reglas Argumentos por omisión

los valores por omisión deberían asignarse en el prototipo de función.

si a cualquier parámetro se le da un valor por omisión en el prototipo de función, a todos los parámetros que siguen también deben asignarles valores por omisión.

si un argumento se omite en la llamada a la función real, entonces todos los argumentos a su derecha también deben omitirse.

E2. caja

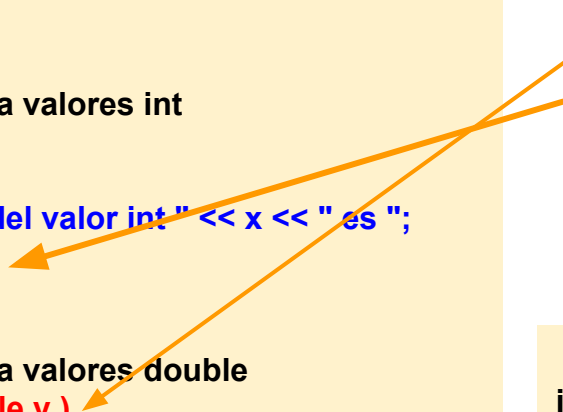
Continuamos con las funciones

```
#include<iostream>

using namespace std;

// función cuadrado para valores int
int cuadrado( int x )
{
    cout << "el cuadrado del valor int " << x << " es ";
    return x * x;
}

// función cuadrado para valores double
double cuadrado( double y )
{
    cout << "el cuadrado del valor double " << y << " es ";
    return y * y;
}
```



Note: no tenemos prototipo

E3. cuadrado

Funciones sobrecargadas

Las funciones sobrecargadas se diferencian mediante sus firmas. Una firma es una combinación del nombre de una función y los tipos de sus parámetros (en orden)

```
int main()
{
    cout << cuadrado( 7 ); // llama a la versión int
    cout << endl;

    cout << cuadrado( 7.5 ); // llama a la versión double
    cout << endl;

    return 0;
}
```

Pila de llamadas a funciones y los registros de activación

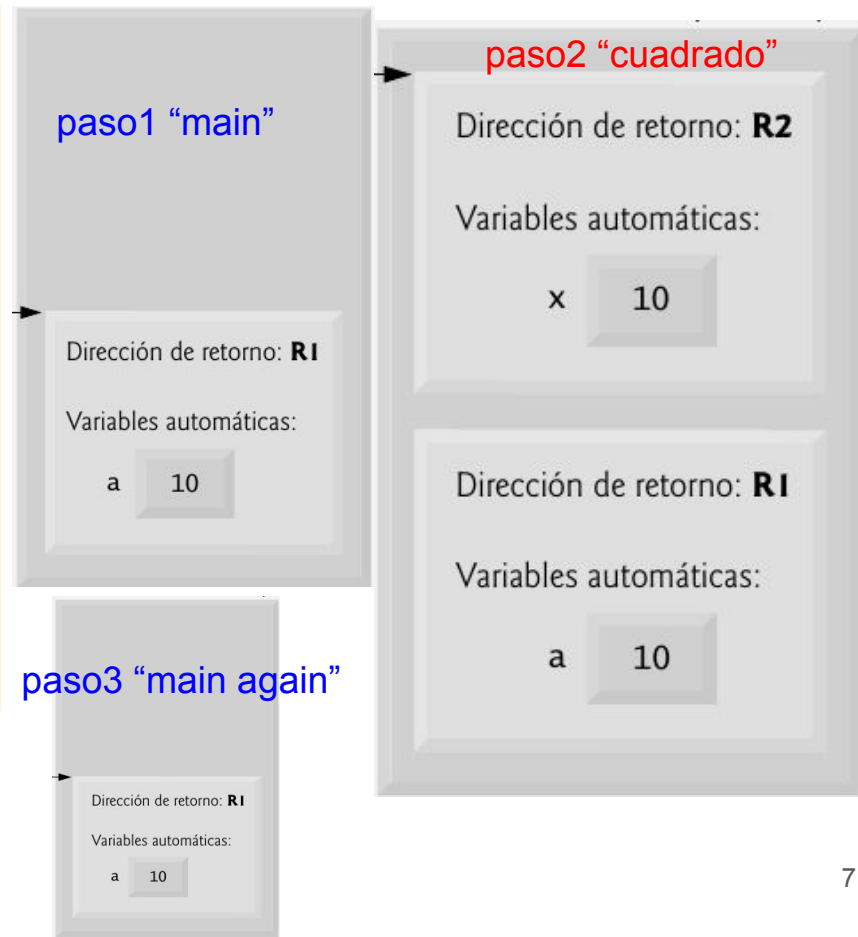


Cuando se coloca un plato en la pila, por lo general se coloca en la parte superior (lo que se conoce como meter el plato en la pila). De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior (lo que se conoce como sacar el plato de la pila). Las pilas se denominan estructuras de datos “último en entrar, primero en salir” (UEPS); el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.



Pila de llamadas a funciones y los registros de activación

```
int cuadrado( int );  
int main()  
{  
    int a = 10;  
    //valor para cuadrado (variable local automática en main)  
  
    cout << a << " al cuadrado: " << cuadrado( a ) << endl;  
    return 0;  
}  
  
int cuadrado( int x ) // x es una variable local  
{  
    return x * x; // calcula el cuadrado y devuelve el resultado  
}
```



```
int cuadradoPorValor( int );  
void cuadradoPorReferencia( int & );
```

```
int main()  
{  
    int x = 2;  
    int z = 4;  
  
    // demuestra cuadradoPorValor  
    cout << "x = " << x << " antes de cuadradoPorValor\n";  
    cout << "Valor devuelto por cuadradoPorValor: "  
        << cuadradoPorValor( x ) << endl;  
    cout << "x = " << x << " despues de cuadradoPorValor\n" << endl;  
  
    // demuestra cuadradoPorReferencia  
    cout << "z = " << z << " antes de cuadradoPorReferencia" << endl;  
    cuadradoPorReferencia( z );  
    cout << "z = " << z << " despues de cuadradoPorReferencia" << endl;  
    return 0;  
}
```

```
int cuadradoPorValor( int numero )  
{  
    return numero *= numero; // no se modificó el argumento de la función que hizo la llamada  
}
```

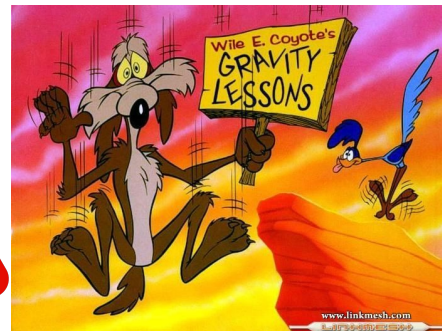
```
void cuadradoPorReferencia( int &refNumero )  
{  
    refNumero *= refNumero; // se modificó el argumento  
}
```

E. porvalor_referencia

Parametros por referencia

Una desventaja del paso por valor es que, si se va a pasar un elemento de datos extenso, el proceso de copiar esos datos puede requerir una cantidad considerable de tiempo de ejecución y espacio en memoria.

El paso por referencia es bueno por cuestiones de rendimiento, ya que puede eliminar la sobrecarga de copiar grandes cantidades de datos en el paso por valor.



El paso por referencia puede debilitar la seguridad, ya que la función a la que se llamó puede corromper los datos de la función que hizo la llamada

Plantillas de funciones “template”

```
template < class T > // o template< typename T >
T maximo( T valor1, T valor2, T valor3 )
{
    T valorMaximo = valor1; // asume que valor1 es maximo

    // determina si valor2 es mayor que valorMaximo
    if ( valor2 > valorMaximo ){valorMaximo = valor2;}

    // determina si valor3 es mayor que valorMaximo
    if ( valor3 > valorMaximo ){valorMaximo = valor3;}

    return valorMaximo;
} // fin de la plantilla de función maximo
```

E5:
panti.h and myplanti

Si no se coloca la palabra clave "class" o "typename" (que son sinónimos) antes de cada parámetro de tipo formal de una plantilla de función (por ejemplo, escribir <class S, T> en vez de <class S, class T>), se produce un error de sintaxis.

Introducción a las clases

```
class nombreClase
{
  public:
    lista_de_funciones_miembro // pueden ser los prototipos o implementación de la función

  private:
    lista_datos_miembro; // son las variables donde defines el tipo de dato nombre variable y ;
};
```

Vamos a empezar con un ejemplo que consiste en la clase “HolaClase”.
Primero vamos a describir cómo definir una clase y una **función miembro**. Después explicaremos cómo se **crea un objeto**, y cómo llamar a una función miembro de éste.

Introducción a las clases

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

```
class HolaClase
```

```
{
public:
    void displayMessage()
    {
        cout << "Bienvenido al mundo de los objetos!" << endl;
    }
};
```

```
int main()
```

```
{
    HolaClase myHolaClase;
    myHolaClase.displayMessage();

    return 0;
}
```

define la clase.

convención: letra mayúscula
inicio del nombre

etiqueta del
especificador de acceso
public:

función miembro.
tenga cuidado es tipo
"**void**". tipo de valor
de retorno.

NO olvide este ";

crea un objeto de la clase

llamada a la función miembro.
operador punto "."

Plantillas de clases “template”

```
class MyHolder {  
public:  
    int first;  
    int second;  
    int third;  
    int sum() {  
        return first + second + third;  
    }  
};
```

```
class AnotherHolder {  
public:  
    float first;  
    float second;  
    float third;  
    float sum() {  
        return first + second + third;  
    }  
};
```

```
template <typename T>  
class CoolHolder {  
public:  
    T first;  
    T second;  
    T third;  
    T sum() {  
        return first + second + third;  
    }  
};
```

E. myclass_Template

Resumen funciones

```
tipo_de_retorno nombre_de_funcion( parametros )  
{  
    //Cuerpo de la funcion  
    return valor_de_retorno;  
}
```

- **tipo_de_retorno:** es el tipo de valor que devuelve la función. Puede ser un tipo de dato básico, como int, float, double, char, etc., o un tipo definido por el usuario, como una clase o estructura.
- **nombre_de_función:** es el nombre que se le da a la función, que se utiliza para llamarla desde el código.
- **parámetros:** son los valores que se pasan a la función para que los utilice en su ejecución. Pueden ser de cualquier tipo de dato, separados por comas, y pueden ser opcionales.
- **cuerpo_de_la_función:** es el conjunto de instrucciones que se ejecutan cuando se llama a la función. Aquí es donde se define la tarea que realiza la función.
- **valor_de_retorno:** es el valor que devuelve la función al final de su ejecución. Este valor es opcional y puede ser de cualquier tipo de dato que coincida con el tipo de retorno de la función

RECUERDE: Hay funciones en C++ que no necesitan devolver un valor, ya que simplemente realizan una tarea y no requieren que se devuelva un resultado. Estas funciones se llaman funciones void

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>
#include <iomanip>

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;
}
```

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."

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;  
}
```

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. Código `example4_modificarA`

```
#include <iostream>
#include <iomanip>

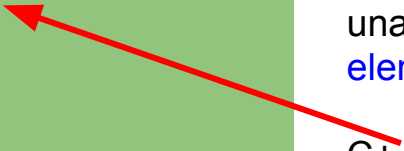
using namespace std;

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. example5_busquedalineal

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.

¿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++ )
```

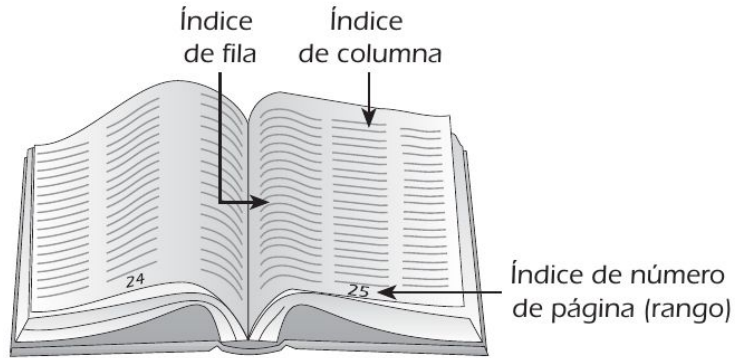
```
    cout << a[ i ][ j ] << ' ';
```

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

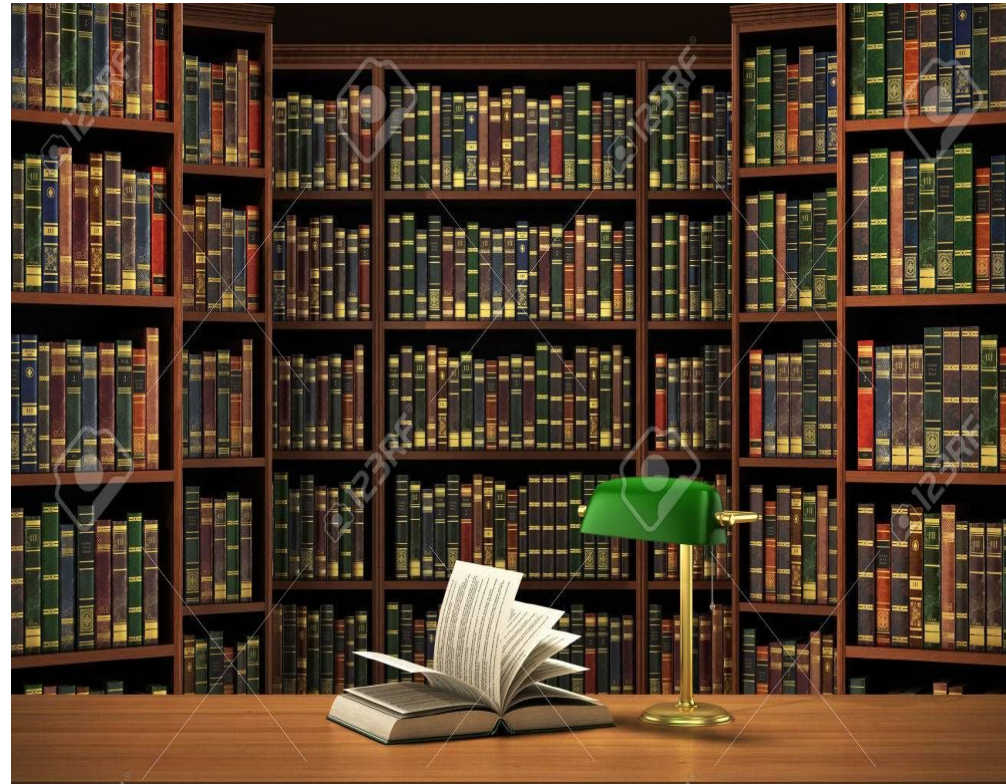
```
}
```

```
}
```

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

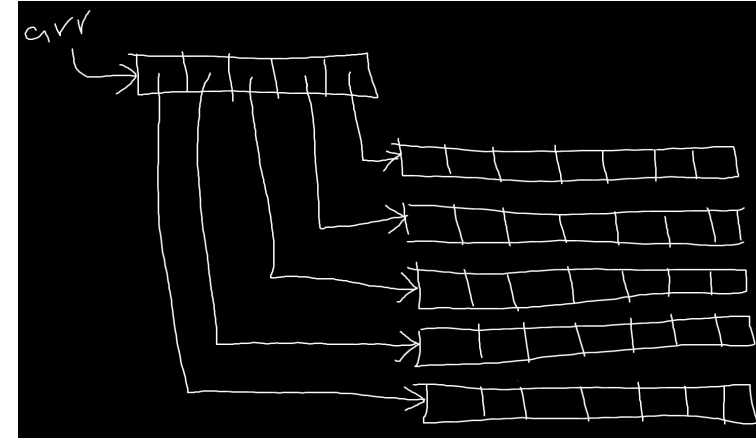
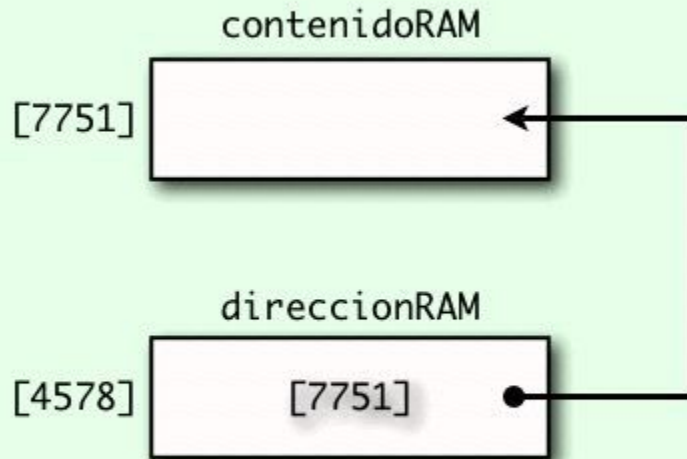
Podemos poner un poco de luz para ver la grandeza de Khazad-dûm



- ¿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

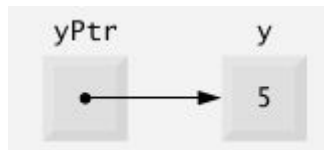


Los apuntadores nos dan acceso al funcionamiento interno de la computadora y la estructura de almacenamiento básico

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 direcci3n 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\nDemostraci3n de que * y & son inversos "  
          << "uno del otro.\n&*aPtr = " << &*aPtr  
          << "\n*&aPtr = " << *&aPtr << endl;  
  
    return 0;  
}
```

Declaraciones e inicialización de variables apuntables

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;
```





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 ?

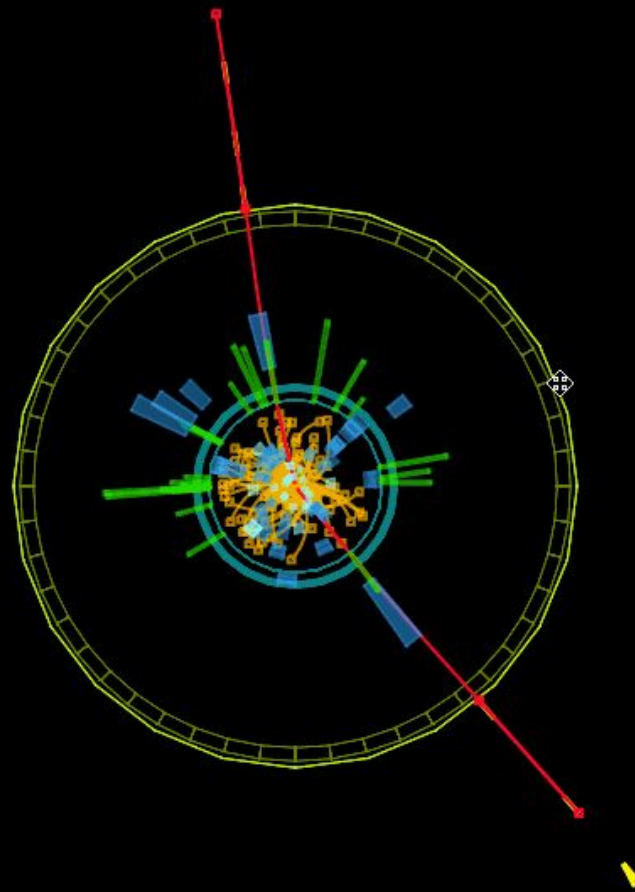
- Tracks (reco.) ☒
- Clusters (Si Pixels) ☐
- Clusters (Si Strips) ☐
- Rec. Hits (Tracking) ☐

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

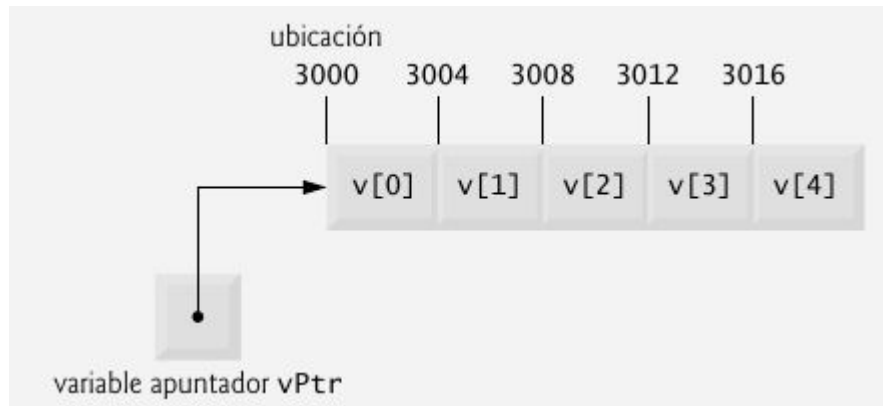
Type Fun (type *, type *) ;

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.



```
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 example3_notacionApun.cpp

```
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 MorePointers



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]`

backup



Errores de manejo de memoria

- Segmentation fault: cuando se trata de acceder una posición de memoria no reservada.
- Double free corruption: cuando se trata de liberar dos veces el mismo puntero
- No liberar memoria asignada a un puntero.
- Pedir una cantidad de memoria que no se puede asignar.

```
#include<iostream>
#include <limits> // std::numeric_limits
using namespace std;
void print(int *p,int n){
    cout<<"[";
    for(int i=0;i<n;i++){
        cout<<p[i]<<" ";
    }
    cout<<"]"<<endl;
}
void llenar_matriz(int **m,int n,int m, int valor){
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            m[i][j]=valor;
        }
    }
}
int main(){
    int *p;
    //caso 1 segfault, la posición 100 del vector p no existe
    //cout<<p[100]<<endl;
    // caso 2 liberar p sin haber sido asignado
    //delete p;
    p=new int[2];
    p[0] = 1;
    p[1] = 1;
    int *t=p;
    print(p,2);
    print(t,2);
    cout<<p<<endl<<t<<endl;
    delete p;
    //delete t; // ¿qué va a pasar?

    //para este ejemplo vamos a asignar mas memoria de la disponible
    //unsigned long int max = std::numeric_limits<unsigned long int>::max();
    //unsigned int max = std::numeric_limits<unsigned int>::max();
    unsigned int max = 5;
    cout<<max<<endl;
    int **m;
    m=new int*[max];
    for(auto i=0;i<max;i++){
        m[i]=new int[max]; //ojo al correr este código te va a bloquear el pc
    }
    return 0;
}
```