

# PROGRAMACIÓN ORIENTADA A OBJETOS USANDO C++

1ª EDICIÓN

2ª SESIÓN

1

Curso 2015-2016

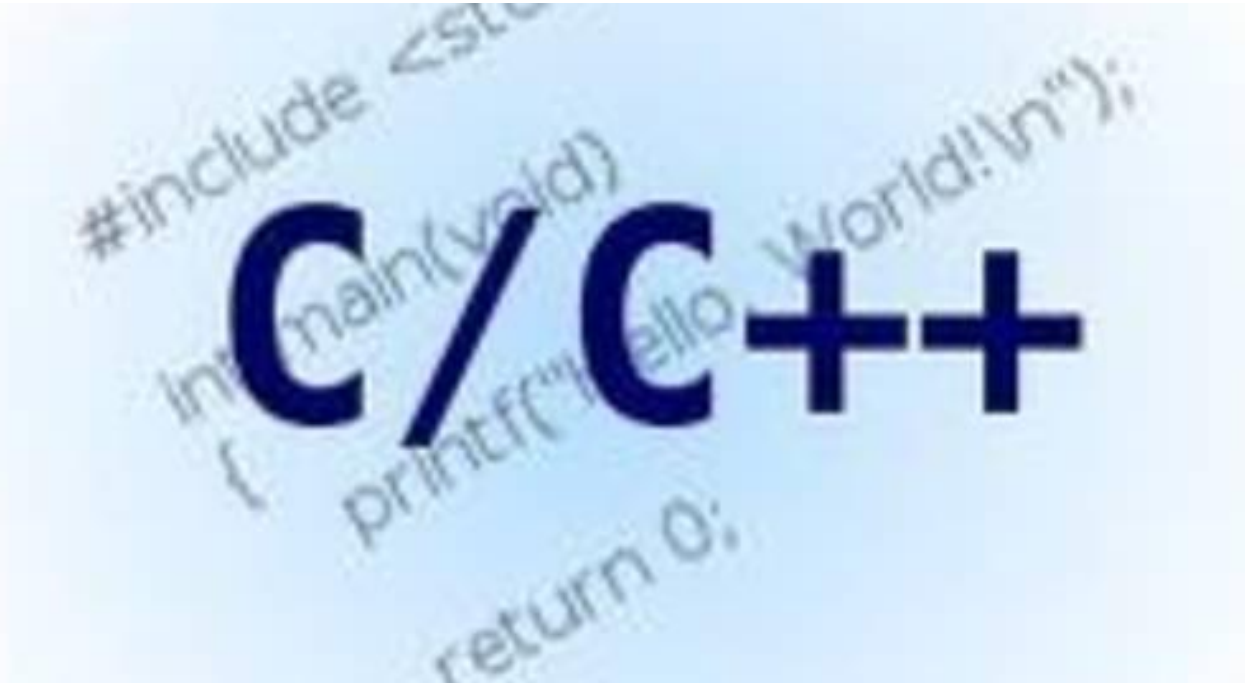
# ÍNDICE

1. Introducción C/C++
2. Entrada/Salida básica
3. Los Strings
4. Punteros
5. Referencias
6. STL
  1. Vectores
  2. Colas
  3. Mapas
  4. Listas

# ÍNDICE DE LA SESIÓN 3

1. Clases y Objetos
2. Herencia
3. Sobrecarga de Operadores
4. Polimorfismo
5. Abstracción
6. Encapsulación
7. Interfaces
8. Otros

## 1- C/C++



# 1- INTRODUCCIÓN C/C++

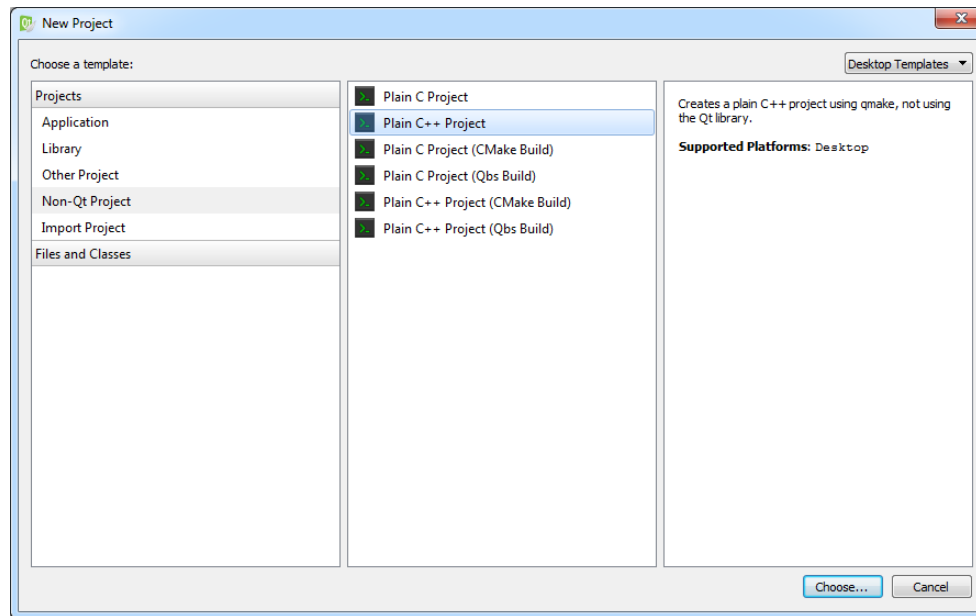
- C++ es un lenguaje de programación diseñado a mediados de los años 80 por Bjarne Stroustrup.
- Es una extensión de C que permite la manipulación de objetos.
- C++ sigue un paradigma híbrido
  - Programación estructurada (C)
  - Programación orientada a objetos (C++)

# 1- INTRODUCCIÓN C/C++

- Siguen un estándar denominado ISO C++
- Actualmente es el estándar más usado y extendido de C++: ISO/IEC C++ 2003
- Por lo tanto nos centraremos en este estándar.
- Finalmente, en la últimas sesiones veremos las mejoras que aporta C++11 y C++14

# 1- INTRODUCCIÓN C/C++

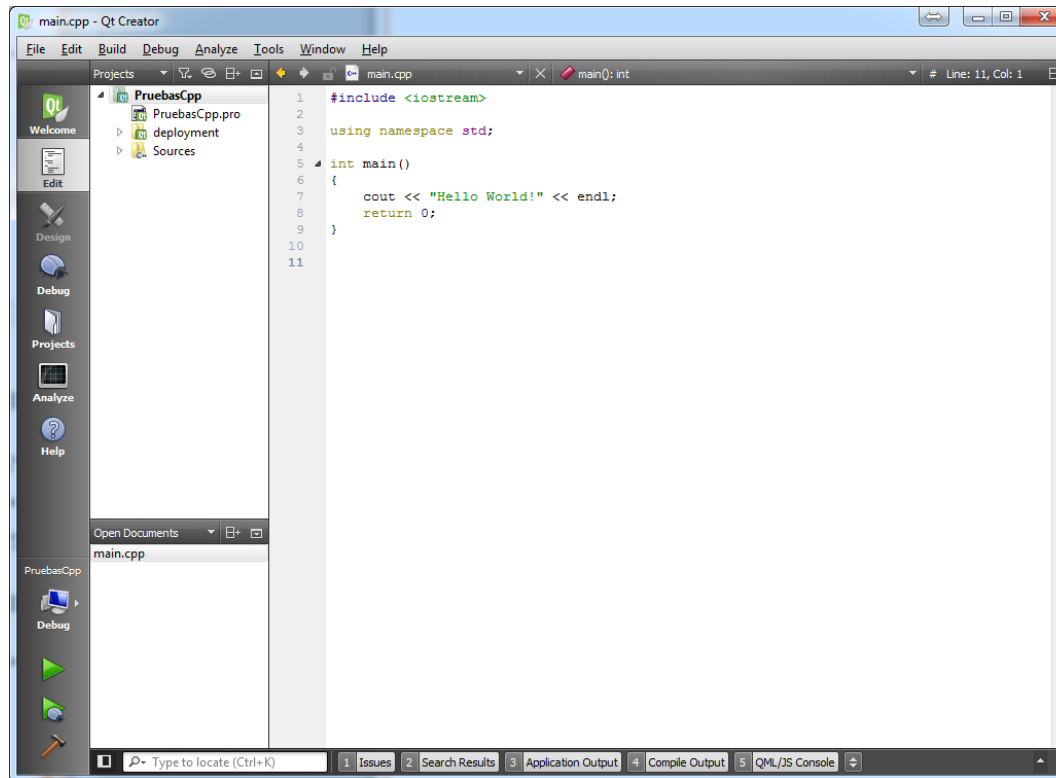
- Vamos a crear un proyecto de C++ en Qt. Vamos a «welcome», Non-Qt Project and Plain C++ Project



- Finalmente elegimos un directorio y un nombre para nuestro proyecto.

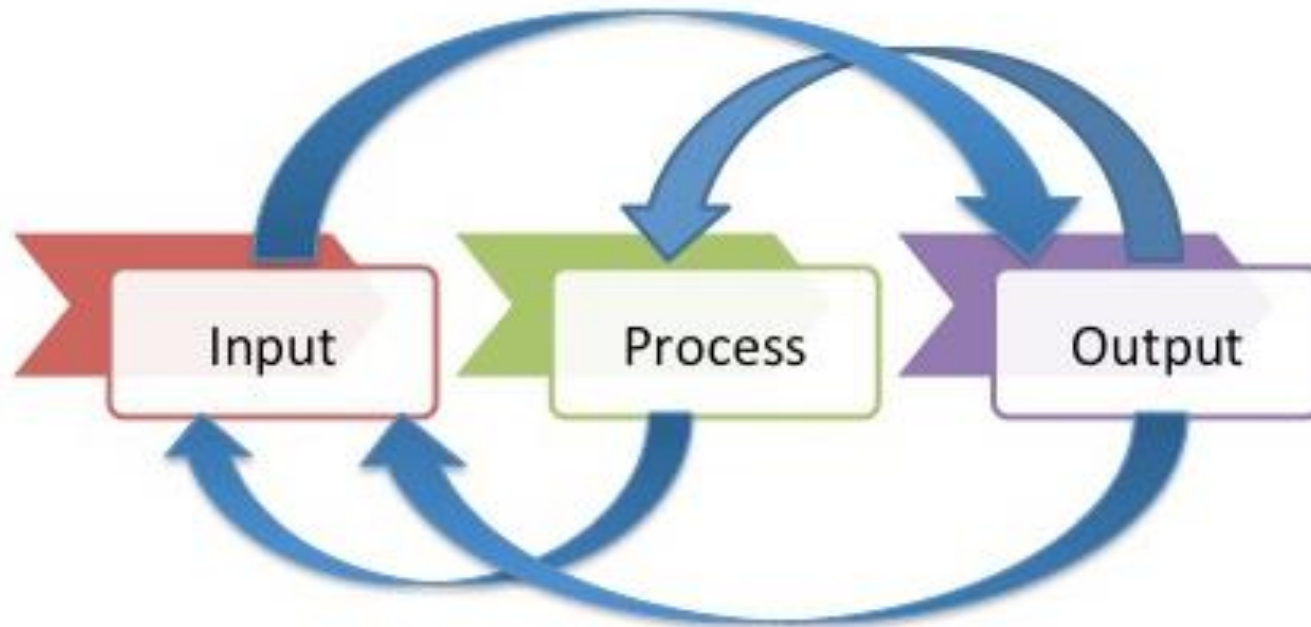
# 1- INTRODUCCIÓN C/C++

- Automáticamente Qt Creator nos crea un «Hola Mundo» en C++





## 2- ENTRADA/SALIDA BÁSICA



## 2- ENTRADA/SALIDA BÁSICA

- La librería estándar de C++ (**std**) provee de una gran cantidad de funciones de entrada y salida (**C++ I/O**).
- En esta sección nos vamos a centrar en las más importantes (**#include <iostream>**).
- Esta librería trabaja con secuencias de Bytes que pueden provenir de un teclado, un disco duro, una conexión a la red, etc.

## 2- ENTRADA/SALIDA BÁSICA

- **cout** → flujo de salida estándar. Con esta función usa la salida estándar de nuestro dispositivo, normalmente la pantalla.
- Esta función se usa así:

```
#include <iostream>
```

```
main(){
```

```
    char cadena[] = "Hola C++";
```

```
    std::cout << "El valor de la cadena es " << cadena << std::endl;
```

```
}
```

## 2- ENTRADA/SALIDA BÁSICA

- La función `cout` está dentro de la librería estándar de C++ (`std`), por lo tanto debemos escribir:
  - *`std::cout` y `std::endl`;*
- Otra alternativa es añadir el espacio de nombre de la librería estándar, al principio de nuestro programa, así:

```
#using namespace std;
```

```
...
```

```
cout << "El valor de la cadena es " << cadena << endl;
```

## 2- ENTRADA/SALIDA BÁSICA

- **cin** → flujo de entrada estándar. Con esta función usa la entrada estándar de nuestro dispositivo, normalmente el teclado.
- Esta función se usa así:

```
#include <iostream>
```

```
...
```

```
char nombre[100];
```

```
std::cin >> nombre;
```

## 2- ENTRADA/SALIDA BÁSICA

- **cerr** → flujo de salida de errores estándar. Con esta función usa la salida estándar de nuestro dispositivo, normalmente la pantalla. No utiliza el buffer de salida, por lo que los mensajes aparecen inmediatamente por pantalla.
- Esta función se usa así:

```
#include <iostream>
```

```
...
```

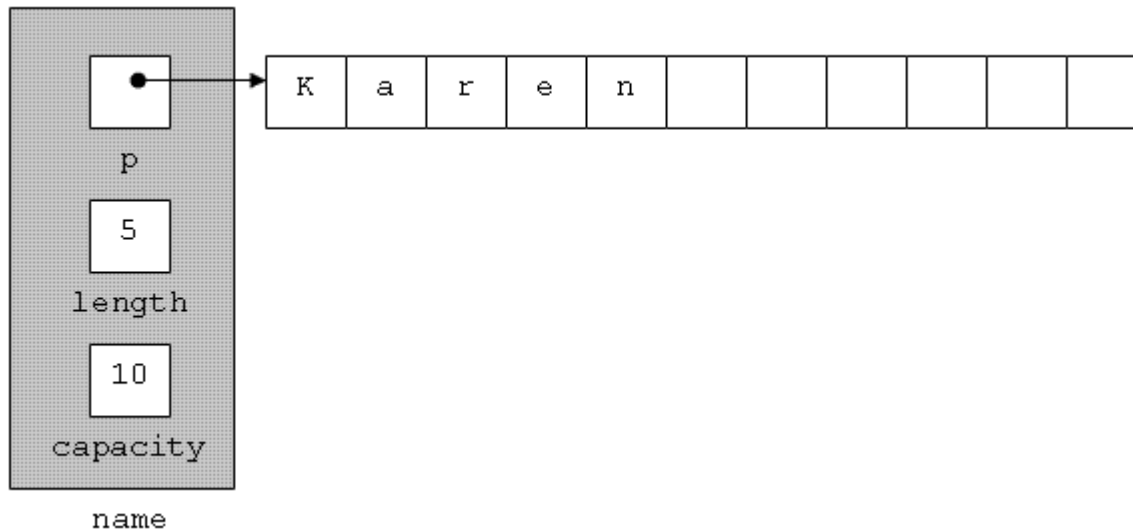
```
char problema[] = " Dispositivo no encontrado...";
```

```
std::cerr << "Error: " << problema << std::endl;
```

## EJERCICIO

- Crear un programa usando las funciones que hemos visto anteriormente para pedir al usuario diferente información y que se vaya mostrando por pantalla. Pedir el nombre, apellidos, edad, etc.
- Usar la salida «**cerr**» para mostrar el siguiente posibles error:
  - si la edad es negativa

# 3-LOS STRINGS





## 3-LOS STRINGS

- La librería estándar de C++ (std) proporciona la clase «**string**» para facilitar el trabajo con cadenas de caracteres.
- Proporciona toda la funcionalidad que teníamos en C con los arrays de char de una forma sencilla e intuitiva.
- Necesitamos añadir al principio de nuestro programa:  
*#include <string>*

## 3-LOS STRINGS

- La declaración se realiza así:

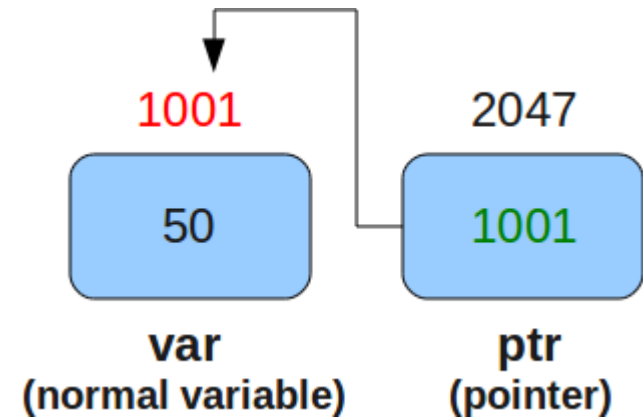
```
string cadena1 = "Hola Mundo...";
```

- *Algunas funciones importantes son:*
  - *cadena1.size()* → devuelve el número de caracteres usados.
  - *cadena1.at(3)* → devuelve el carácter de la tercera posición (empieza en 0). Similar a *cadena1[3]*
  - *string res = cadena1 + cadena2* → concatenación con el "+"
  - *cadena1.erase(1, 5)* → elimina los caracteres desde la posición 1 hasta 5 más adelante.
  - *cadena1.c\_str()* → convierte el string en un array de char

# EJERCICIO

- A partir el ejercicio anterior cambiar las partes necesarias para usar los Strings y almacenar las cadenas de caracteres.
- Añade que el usuario introduzca también su dni
- Usar la salida «**cerr**» para mostrar los posibles errores:
  - El nombre o los apellidos contienen un número
  - Último carácter del dni es un número y no una letra
- Mostrar el número de caracteres que tiene el nombre y los apellidos. Si tiene más de 6 caracteres el nombre recortarlo para que se quede con esta longitud máxima.

## 4- PUNTEROS



## 4- PUNTEROS

- ¿Qué es un puntero?
  - Un puntero es una variable que almacena la dirección de memoria de otra variable.
  - Los punteros permiten manejar variables, estructuras y arrays de una forma muy eficiente.
  - Los punteros permiten reservar memoria dinámica, es decir, mientras se está ejecutando el programa.

## 4- PUNTEROS

- ¿Qué es un puntero?
  - Con el operador & obtenemos la dirección de memoria
  - Con el operador \* obtenemos el contenido de la variable a la que apunta.

## 4- PUNTEROS

- Ejemplo, ¿qué imprime este código?

```
int var1;
```

```
char var2[10];
```

```
cout << "Address of var1 variable: "; cout << &var1 << endl;
```

```
cout << "Address of var2 variable: "; cout << &var2 << endl;
```

## 4- PUNTEROS

- Definición de un puntero:

- `tipo_de_dato *nombre_puntero;`

- Ejemplos:

- `int *puntero_i;`
- `double *puntero_d;`
- `float *puntero_f;`
- `Coche *puntero_coche;`



## 4- PUNTEROS

- Ejemplo usando un puntero:

```
int var = 20;          // actual variable declarada.
```

```
int *ip;              // variable puntero
```

```
ip = &var;            // almacena la dirección de la variable
```

```
cout << "El valor de la variable es: "; cout << var << endl;
```

```
cout << "La direccion de memoria de la variable es: "; cout << ip << endl;
```

```
cout << "El valor del puntero es: "; cout << *ip << endl;
```

## 4- PUNTEROS

- Es posible asignar un NULL al puntero cuando no sabemos a que variable va a apuntar.
  - El valor del puntero será 0
  - Ejemplo:

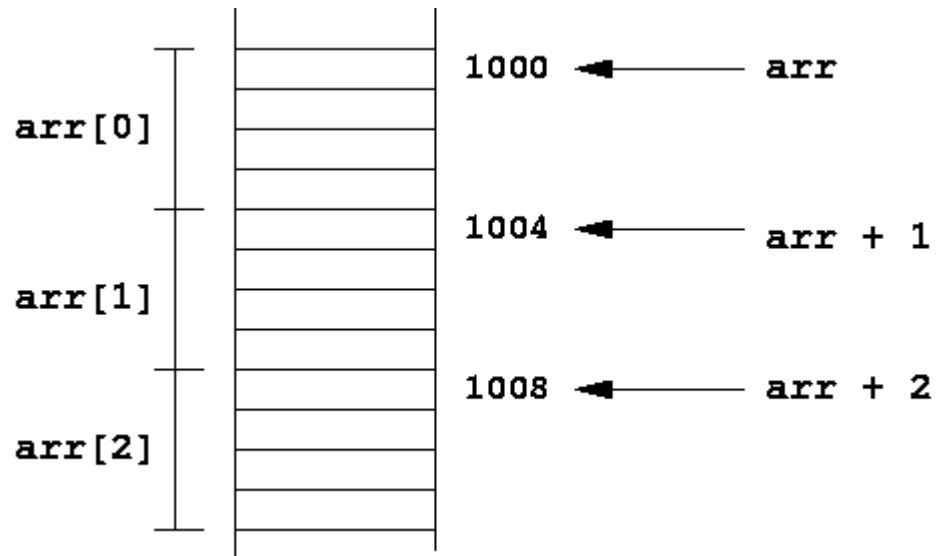
```
int *ptr = NULL;  
cout << "El valor del puntero es " << ptr ;
```

- *Se puede comprobar el contenido de un puntero NULL así:*

```
if(ptr) // true si el puntero es no NULL  
if(!ptr) // true si el puntero es NULL
```

## 4- PUNTEROS

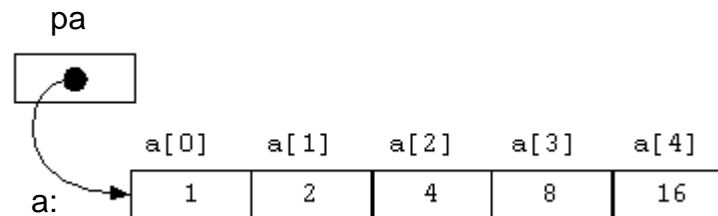
- Otro uso muy importante de los punteros es el manejo de arrays usando la aritmética de punteros.



## 4- PUNTEROS

- Como hemos visto anteriormente un puntero es una variable por lo tanto podemos realizar operaciones aritméticas con ella.
- Para entender este concepto vamos a suponer que tenemos un puntero (pa) de enteros (int de 4 Bytes). ¿Cuál es el resultado de estas operaciones?

- `pa+1`
- `pa+3`
- `*(pa++)`
- `*(pa+1)`
- `*(pa+3)`
- `(*pa+1)`
- `(*pa+3)`



## 4- PUNTEROS

```
int a[5]={1,2,4,8,16};
```

```
int *pa;
```

```
pa=&a[0];
```

```
cout << "El valor de pa debe ser el valor de &a[0] " << pa <<endl ;
```

```
cout << "El valor de &a[0] debe ser el valor de pa " << &a[0] <<endl ;
```

```
cout << "El valor de pa+ + es: " << (pa+ +) <<endl ;
```

```
cout << "El valor de pa + 1 es: " << (pa + 1) <<endl ;
```

```
cout << "El valor de pa + 3 es: " << (pa + 3) <<endl ;
```

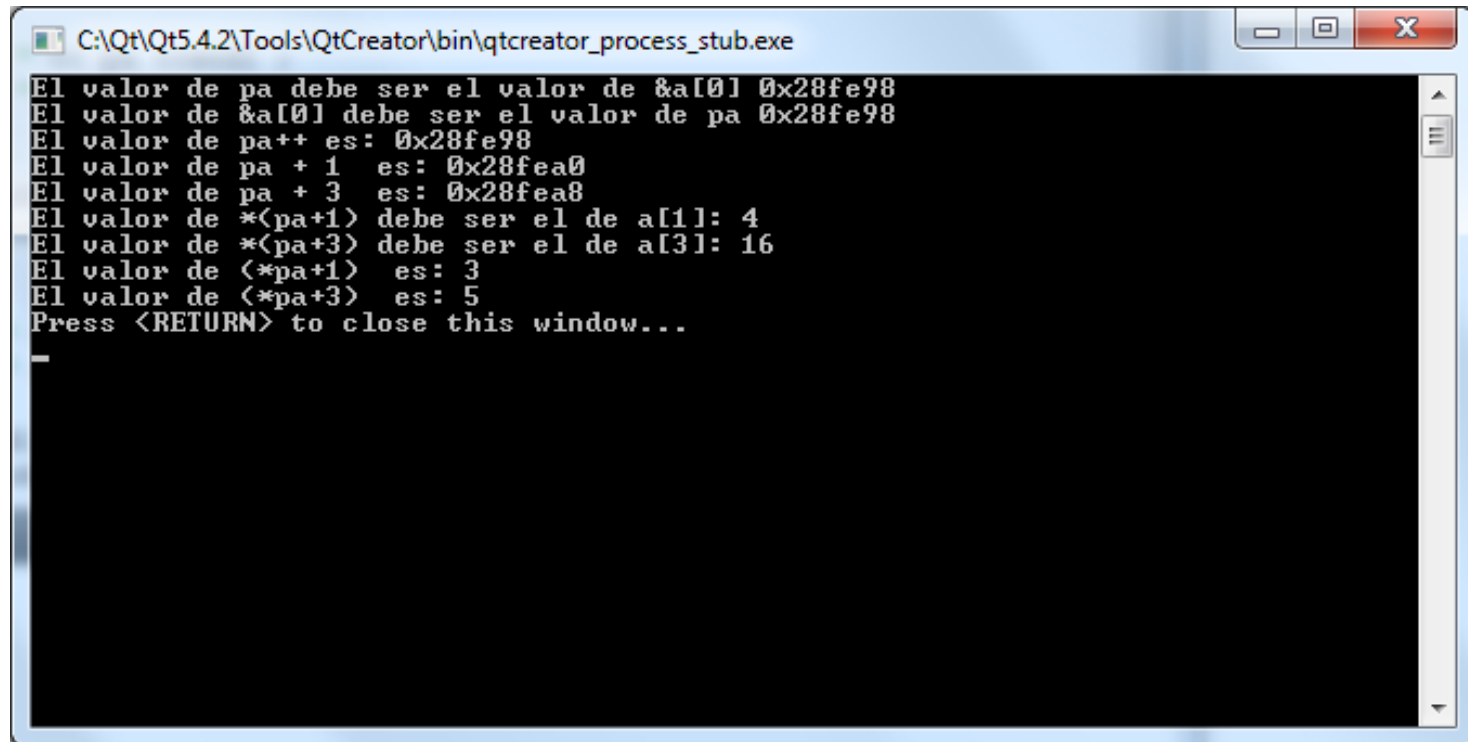
```
cout << "El valor de *(pa+1) debe ser el de a[1]: " << *(pa+1)<<endl ;
```

```
cout << "El valor de *(pa+3) debe ser el de a[3]: " << *(pa+3) <<endl ;
```

```
cout << "El valor de (*pa+1) es: " << (*pa+1)<<endl ;
```

```
cout << "El valor de (*pa+3) es: " << (*pa+3) <<endl ;
```

## 4- PUNTEROS



```
C:\Qt\Qt5.4.2\Tools\QtCreator\bin\qtcreator_process_stub.exe
El valor de pa debe ser el valor de &a[0] 0x28fe98
El valor de &a[0] debe ser el valor de pa 0x28fe98
El valor de pa++ es: 0x28fe98
El valor de pa + 1 es: 0x28fea0
El valor de pa + 3 es: 0x28fea8
El valor de *(pa+1) debe ser el de a[1]: 4
El valor de *(pa+3) debe ser el de a[3]: 16
El valor de (*pa+1) es: 3
El valor de (*pa+3) es: 5
Press <RETURN> to close this window...
-
```

## 4- PUNTEROS

- Paso de argumentos con punteros a una función.
  - Es muy rápido ya que solo pasamos una dirección de memoria
  - Si modificamos la variable también quedará modificada fuera (paso de argumentos por referencia)
- Ejemplo:


```
int vector[3] = {10, 15, 20};  
int mi_variable = 8;  
int *mi_puntero = &mi_variable;
```

...

```
void modificaVariable(int *entrada){  
    *entrada = 0;  
}
```

```
modificaVariable( vector )  
modificaVariable( &mi_variable )  
modificaVariable( mi_variable )  
modificaVariable( mi_puntero )
```

¿Son  
correctos?



Espero una  
dirección de  
memoria...

## 4- PUNTEROS

- Retorno de argumentos con punteros de una función.
  - Es muy rápido ya que solo pasamos una dirección de memoria
  - Podemos devolver más de un argumento
- Ejemplo:

```
int *p;  
p = obtenerNumeros();  
cout << *p << endl;  
cout << *(++p) << endl;  
cout << *(++p) << endl;  
...
```

```
int* obtenerNumeros( void ){  
    static int vec[3] = {5, 10, 15};  
    return vec;  
}
```

¿Qué pasa si quitamos el  
static en  
obtenerNumeros() ?



## EJERCICIO

- Crear una función que a partir de dos números (pasados con punteros) calcule la suma, el producto y el número máximo y los devuelva.
  - (Nota) Utilizar los punteros para devolver estos tres resultados.
- Mostrar los resultados en la función main

## 5- REFERENCIAS

- Una variable referencia es un alias, es decir, otro nombre para una variable existente.
- La referencia debe ser inicializada con una variable. Después se podrá hacer uso de esta variable de una forma sencilla, como si fuera una variable normal.
- Las referencias modifican el contenido de la variable.
- No se pueden tener referencias a NULL.
- No se puede cambiar la referencia a otra variable una vez inicializada.

## 5- REFERENCIAS

- Ejemplo:

```
int var = 25;
```

```
int& ref = var;
```

```
ref = 20;
```

```
cout << ref << endl;
```

```
cout << var << endl;
```

¿Cuál es la salida?

## 5- REFERENCIAS

- Paso de argumentos con referencias a una función.
  - Es muy rápido ya que solo pasamos una dirección de memoria
  - Si modificamos la variable también quedará modificada fuera (paso de argumentos por referencia)
  - Es más sencillo de trabajar con referencias que con punteros.
- Ejemplo:

```
int mi_variable = 8;  
int &mi_referencia = mi_variable;  
modificaVariable( mi_variable );           ó           modificaVariable( mi_referencia );  
  
cout << mi_variable << endl;  
  
void modificaVariable(int &entrada){  
    entrada = 0;  
}
```

## 5- REFERENCIAS

- Retorno de argumentos con referencias desde una función.
  - Es muy rápido ya que solo pasamos una dirección de memoria
  - Podemos devolver más de un argumento
- Ejemplo:

```
int vec[3] = {5, 20, 25};
```

```
int& setValor(int index){  
    return vec[index];  
}
```

¿Cuál es la salida de este código?

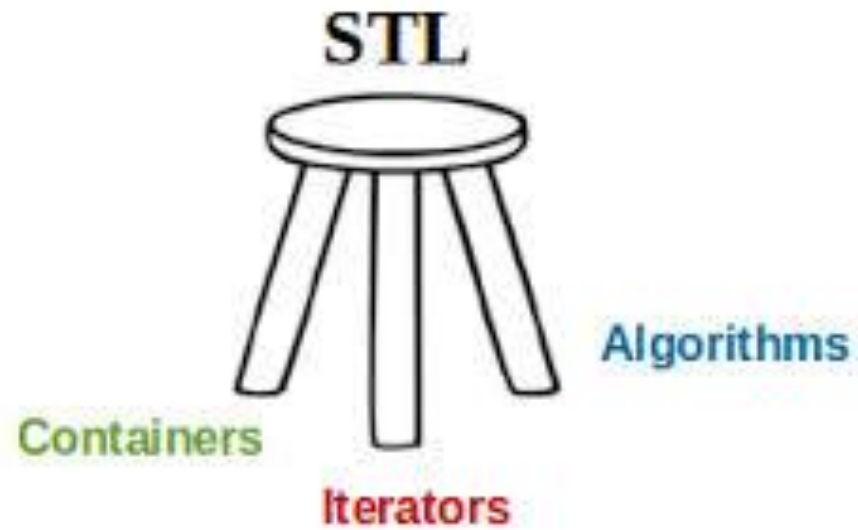
```
setValor(0) = 15;  
setValor(1) = 99;
```

```
cout << vec[0] << " " << vec[1] << " " << vec[2] << endl;
```

# EJERCICIO

- Utilizando el ejercicio anterior, cambiar los punteros por referencias para pasar los argumentos.
  - ¿Qué opción es mejor para este caso?
- Devolver los argumentos usando referencias.
  - ¿Qué opción es mejor para este caso?

## 6- STL



## 6- STL

- La STL de C/C++ es una librería de plantillas estándar (STL), es un conjunto de clases muy potente de uso general.
- La STL implementa las estructuras de datos más comunes como son vectores, listas, colas, pilas, mapas, etc.

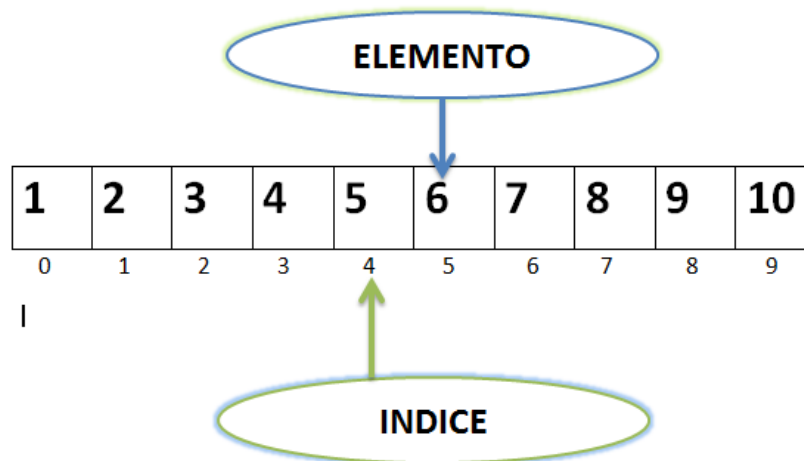


## 6- STL

- La STL sigue esta estructura:
  - Los **contenedores** (Containers) son usadas para manejar colecciones de objetos de un determinado tipo. Existen diferentes tipos de contenedores como las colas, vectores, pilas, etc.
  - Las **operaciones** (Algorithms) actúan sobre los contenedores. Realizan la inicialización, búsqueda, ordenación, etc.
  - Los **iteradores** (Iterators) son usados para navegar dentro de los contenedores de una forma eficiente.

## 6.1- VECTORES

- Los vectores son una secuencia de elementos, como los Arrays, pero pueden cambiar su tamaño dinámicamente.



## 6.1- VECTORES

- Los vectores, al igual que los arrays, almacenan los datos en memoria continuamente. Estos datos pueden ser accesible con un índice o con un puntero. De una forma tan eficiente como en los Arrays.
- Si el tamaño del vector crece, automáticamente cuando se necesita, se reservará más memoria y se copiará el contenido. Esta operación es relativamente lenta.
  - Solución: si sabemos el tamaño podemos reservar el tamaño que necesitamos.

## 6.1- VECTORES

- Declaración e inicialización de los vectores.
  - Necesitamos añadir: `#include <vector>`
- Ejemplos para inicializar un vector:

```
vector<int> v1;  
v1.resize(5);  
for(int i=0; i<5; i++) v1[i] = 10;
```

```
vector<int> v2(5, 10);
```

```
vector<vector<int> > > v3(5, vector<int>(6, 10));
```

## 6.1- VECTORES

- Medir el tamaño, reajustar el tamaño y medir la capacidad.
- *Ejemplos:*

```
vector<int> v1;  
v1.resize(10);  
cout << v1.size() << endl;  
cout << v1.capacity() << endl;  
v1.push_back(5);  
cout << v1.capacity() << endl;  
cout << v1.max_size() << endl;  
cout << v1.empty() << endl;
```

¿Cuál es la salida de este código?

## 6.1- VECTORES

- Operaciones importantes con los vectores:
  - Añadir un elemento al final: *v.push\_back(elemento)*
  - Eliminar el último elemento: *v.pop\_back()*
  - Acceder a un elemento: *v.at( índice )*  
*v[ índice ]*
  - Borrar el contenido: *v.clear()*

## 6.1- VECTORES

- Operaciones importantes con los vectores:


- Añadir elementos de un vector en otro:

```
v.insert(v.begin()+pos_v, v2[pos_v2]);
```


- *Eliminar elementos:*

```
v.erase( v.end()-5, v.end() );
```

Inserta en la posición pos\_v de v el vector v2, el resto de elementos se desplazan.



Elimina los 5 últimos elementos



## 6.1- VECTORES

- Recorrer el contenido de un vector (tres opciones):

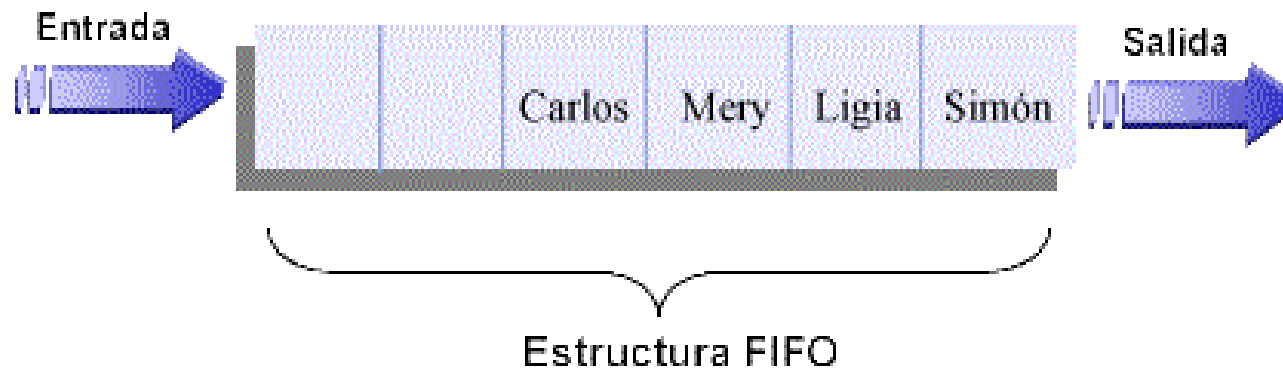
- *for(int i=0; i<v.size(); i++) cout << v[i] << endl;*
- *for(int i=0; i<v.size(); i++) cout << v.at(i) << endl;*
- *vector<int>::iterator it = v.begin();*  
*while( it != v.end() ){*  
*cout << \*it << endl;*  
*it++;*  
*}*



# EJERCICIO

- Crear un vector (vec1) de 20 enteros con valor 1
- Crear otro vector (vec2) de 10 enteros con valores 9, 8, 7, 6,...,0
- Eliminar los tres primeros elementos de vec2
- Eliminar los dos últimos elementos de vec1
- Añadir el primer elemento de vec2 al final de vec1
- Imprimir todos los elementos de ambos vectores
- (Un poco más avanzado) Ordenar ascendentemente vec2
  - Usando la función `std::sort(it_inicio, it_fin)`
  - Añade `#include<algorithm>`

## 6.2- COLAS



## 6.2- COLAS

- La cola (queue) es un contenedor de datos de la STL de C++.
- Estas colas son del tipo FIFO, es decir, primero en entrar primero en salir.
- Necesitamos añadir: `#include<queue>`

## 6.2- COLAS

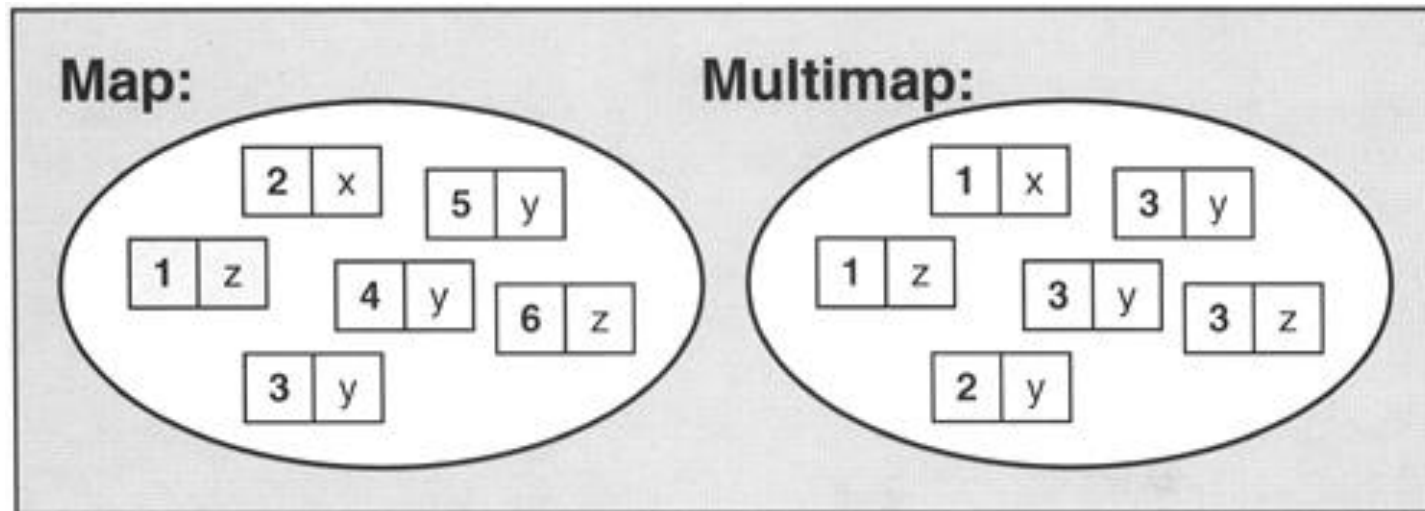
### ○ Operaciones con colas:

- Crear una cola: `queue<int> cola;`
- Comprobar si esta vacía: `cola.empty();`
- Añadir un elemento: `cola.push( elemento );`
- Sacar un elemento: `cola.pop();`
- Ver último elemento que ha entrado: `cola.back();`
- Ver elemento que va a salir: `cola.front();`

## EJERCICIO

- Crear una cola para gestionar el orden en el que llegan los alumnos a hacer su matrícula.
  - (Nota) Usando la estructura queue con strings
- Llegan tres alumnos en este orden: Antonio, Eva y Javier
- Se atiende a los dos primeros alumnos
- Llegan María y Carmen
- Se atiende un alumno
- Finalmente llegan Juanjo, Mario y Cristina
- Mostrar los alumnos que hay ahora finalmente en la cola en el orden que van a ser atendidos

## 6.3- MAPAS



## 6.3- MAPAS

- El mapa (map) es un contenedor de datos asociativo formado por la combinación de una clave y un contenido.
- La clave debe ser única, es decir, no puede estar repetida.
- Se suele usar la clave para ordenar el mapa.

## 6.3- MAPAS

- Los mapas permiten el acceso individual a partir de la clave.
- Es necesario añadir: `#include <map>`



## 6.3- MAPAS

- Las operaciones más importantes son:
  - Declaración de un mapa: `map<int,string> mapa;`
  - Añadir elementos: `mapa[0] = "Pedro";`  
`mapa[1] = 'Dani';`  
`mapa.insert(pair<int,string>(2,"Eva"));`
  - Consular elementos: `mapa[0]`  
`mapa.at(1)`
  - Comprobar si está vacío el mapa: `mapa.empty();`

## 6.3- MAPAS

- Las operaciones más importantes son:
  - Obtener el tamaño del mapa: *mapa.size()*;
  - Eliminar un elemento: *mapa.erase( clave )*;
  - Comprobar si existe el elemento con una clave determinada: *mapa.count( clave )*;
  - Eliminar todo el contenido de un mapa: *mapa.clear()*;

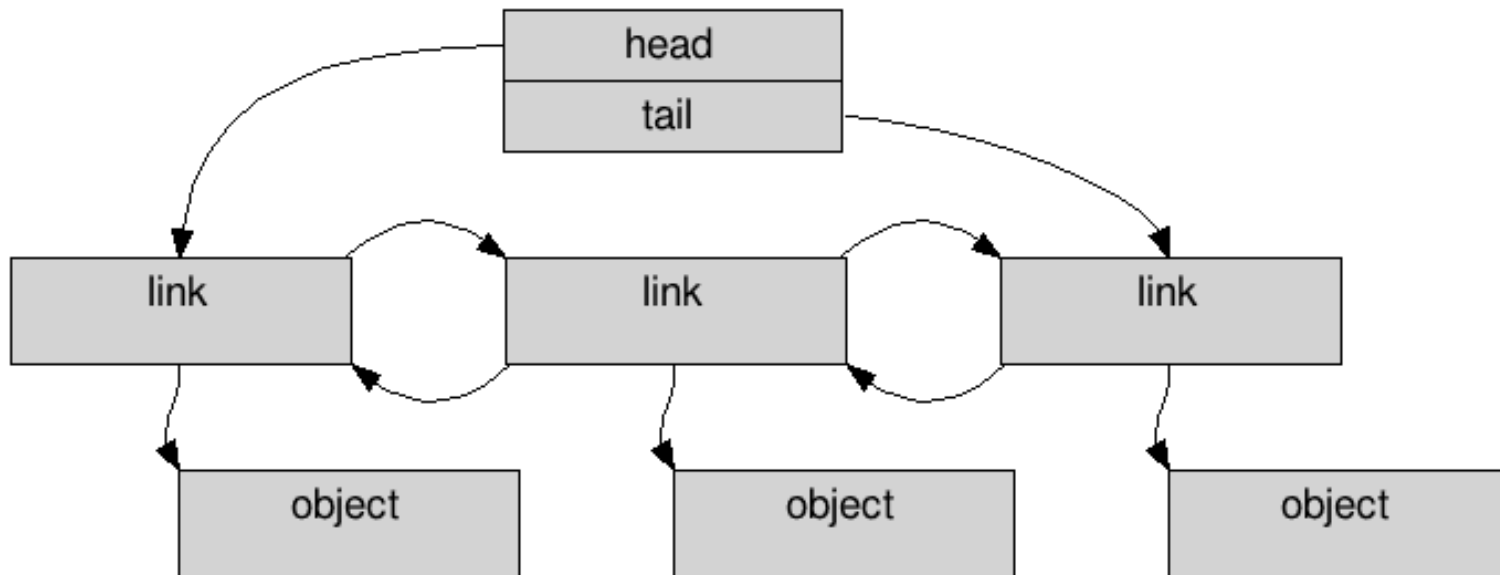
## EJERCICIO

- Crea un mapa de usuarios con los siguientes datos:

Clave	Contenido
100	Antonio Fernández
150	Guillermo Gutiérrez
160	Pedro Garrido
180	Sergio Ruiz

- Eliminar el usuario 160
- Añadir el usuario 220, "Felipe Garrido"
- Mostrar todo el contenido del mapa y contar el número de usuarios

## 6.4- LISTAS



## 6.4- LISTAS

- La lista (list) es un contenedor de datos que contiene una secuencia de elementos.
- Las listas permiten añadir y eliminar elementos en cualquier posición.
- También permite recorrer (iterar) en ambas direcciones.

## 6.4- LISTAS

- Las listas son implementadas como «double-linked lists», es decir, permite recorrer de una forma eficiente la lista en ambas direcciones.
- El contenido de la lista se almacenan el cualquier orden, es decir, no están en posiciones continuas de memoria.
- El orden se consigue, con la lista, porque están enlazados los elementos.

## 6.4- LISTAS

- Las listas son más eficientes (que los arrays, vectores y colas) al insertar, extraer y mover elementos en posiciones aleatorias.
- Son muy útiles para algoritmos de ordenación.
- Es necesario añadir: *#include <list>*

## 6.4- LISTAS

### ○ Operaciones con listas:

- Crear una lista: `list<int> lista;`
- Comprobar si esta vacía: `lista.empty( );`
- Añadir por el principio: `lista.push_front( nuevo );`
- Añadir por el final: `lista.push_back( nuevo );`
- Sacar un elemento del principio: `lista.pop_front( );`
- Sacar un elemento del final: `lista.pop_back( );`



## 6.4- LISTAS

- Operaciones con listas:
  - Obtener el primer elemento: *lista.front( );*
  - Obtener el último elemento: *lista.back( );*
  - *Obtener el tamaño de la lista: lista.size( );*
  - *Limpiar el contenido: lista.clear( );*

## 6.4- LISTAS

- Operaciones con listas:

- Recorrer/modificar una lista, necesitamos los iteradores:

```
list<int>::iterator i;  
for(i=lista.begin(); i != lista.end(); i++)  
cout << *i << endl;
```

- Insertar elementos dentro de la lista, también necesitamos los iteradores:

```
list<int>::iterator i2;  
i2 = lista.begin( );  
i2++; i2++;  
lista.insert(i2, 2 , 10); // En la posición 2, inserta 2 veces el 10
```

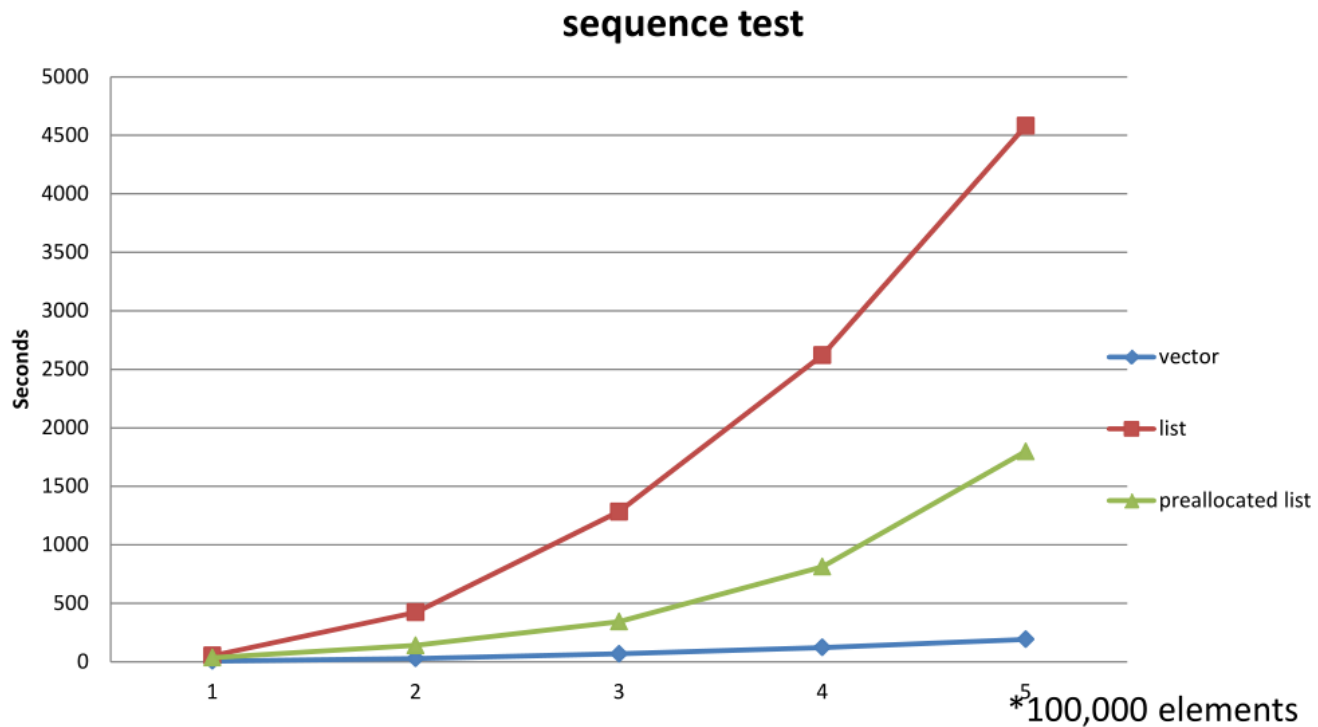
## 6.4- LISTAS

- Operaciones con listas:
  - Eliminar un elemento de la lista, volvemos a necesitar el iterador:  
*lista.erase( i2 );*
  - Eliminar todos los elementos con un mismo valor:  
*lista.remove( 5 );*
  - Ordenar la lista: *lista.sort( );*

## EJERCICIO

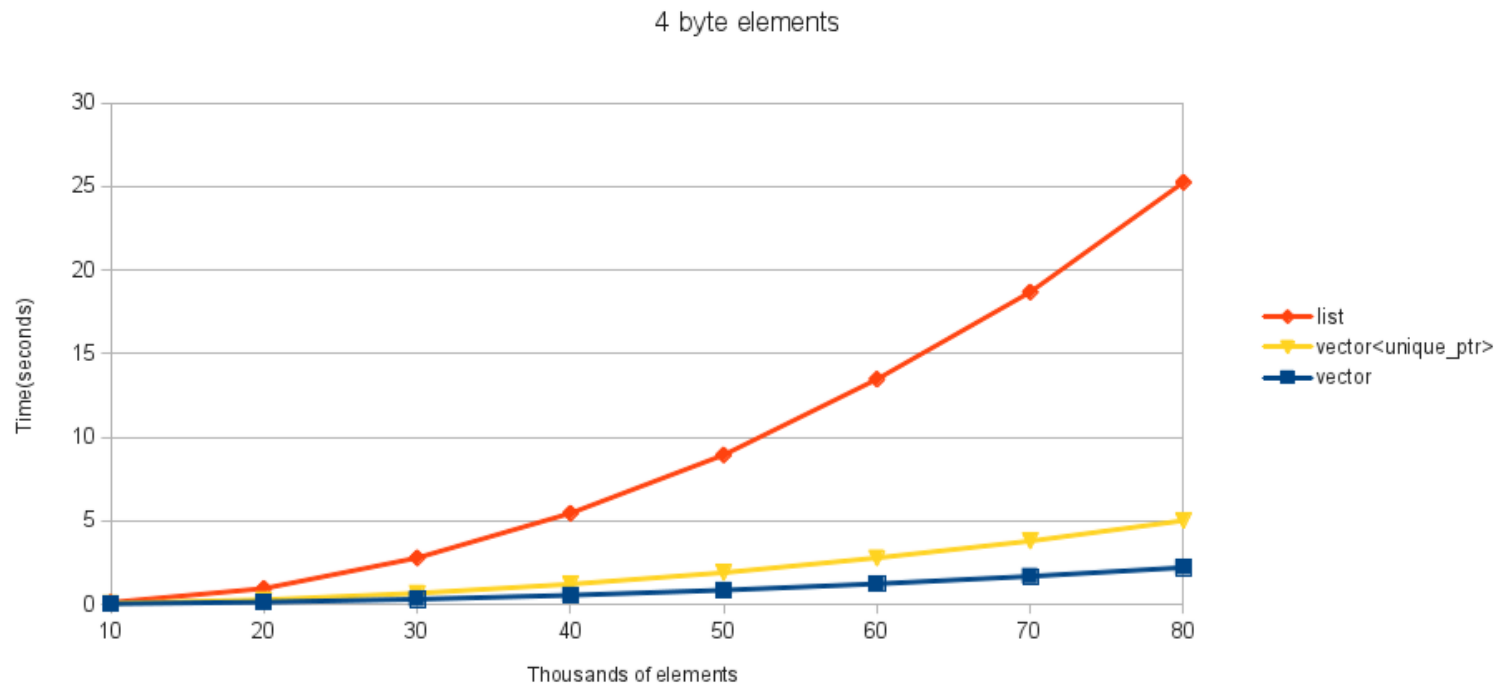
- Crear un lista (list) y añadir los siguientes elementos en este mismo orden:
  - *Por el principio: 15, 10, 8, 6 y 25.*
  - *Por el final: 14, 13, 10, 1, 22 y 12.*
- *Visualizar la lista actual.*
- *Insertar en la posición 2 un 99.*
- *Elimina los dos últimos elementos de la lista.*
- *Elimina el tercer elemento desde el principio.*
- *Elimina todos los 10 de la lista.*
- *Ordenar la lista.*
- *Mostrar el contenido final de la lista.*

## 6-STL. COMPARACIONES



Para insertar y eliminar elementos

## 6-STL. COMPARACIONES



Para insertar y eliminar elementos

## 6-STL. COMPARACIONES

