# Dopando el lenguaje C++ con las bibliotecas Boost

José Tomás Tocino García

Universidad de Cádiz

30 de junio de 2010

# Índice

# Disclaimers iniciales

## Sobre ti, estimado miembro de la audiencia

- Deberías saber algo de C++.
- Deberías saber hacer Makefiles o a compilar en la línea de comandos.

## Sobre mi, oh todopoderoso ponente

- No soy un experto en Boost, ni mucho menos.
- No soy un experto programador, ese derecho solo lo tiene Gerardo.
- Este es mi primer taller, *so bear with me*.

## Total disclaimer 1.0

Según la Wikipedia y el dRAE:

$$\text{librería} \simeq \text{biblioteca}$$

en este contexto.

# Disclaimers iniciales

## Sobre ti, estimado miembro de la audiencia

- Deberías saber algo de C++.
- Deberías saber hacer Makefiles o a compilar en la línea de comandos.

## Sobre mi, oh todopoderoso ponente

- No soy un experto en Boost, ni mucho menos.
- No soy un experto programador, ese derecho solo lo tiene Gerardo.
- Este es mi primer taller, *so bear with me*.

## Total disclaimer 1.0

Según la Wikipedia y el dRAE:

$$\text{librería} \simeq \text{biblioteca}$$

en este contexto.

# Disclaimers iniciales

## Sobre ti, estimado miembro de la audiencia

- Deberías saber algo de C++.
- Deberías saber hacer Makefiles o a compilar en la línea de comandos.

## Sobre mi, oh todopoderoso ponente

- No soy un experto en Boost, ni mucho menos.
- No soy un experto programador, ese derecho solo lo tiene Gerardo.
- Este es mi primer taller, *so bear with me*.

## Total disclaimer 1.0

Según la Wikipedia y el dRAE:

$$\text{librería} \simeq \text{biblioteca}$$

en este contexto.

# ¿Qué es Boost?

Conjunto de bibliotecas de código abierto y revisión por pares para C++.

- Pensadas para la productividad.
- Más de 80 bibliotecas.
- Escritas por los mejores programadores C++. Revisión por pares.
- Nueva versión cada tres meses.
- 12 de estas librerías forman parte del C++ Technical Report 1: Array, Bind, Function, Hash, mem_fn, Random, Ref, Regex, Smart Pointers, Tuple, Type Traits y Unordered.

# Compilando con Boost

## ¿Cómo compilo Boost?

Bueno, en realidad... en la mayoría de los casos no hay **nada que compilar**. La inmensa mayoría de librerías de boost se alojan en ficheros de cabecera que no hace falta compilar. Simplmente se incluyen en tu proyecto.

## Algunas excepciones

Hay bibliotecas de boost que por su complejidad necesitan ser compiladas y enlazadas:

- Boost.Filesystem
- Boost.IOStreams
- Boost.ProgramOptions
- Boost.Python
- Boost.Regex
- Boost.Serialization
- Boost.Signals
- Boost.System
- Boost.Thread
- Boost.Wave

# Organización del taller

Este taller se va a organizar en dos partes:

1. En la primera parte vamos a hacer un repaso por las librerías que forman parte de TR1, explicando brevemente qué hace cada una y poniendo un ejemplo. Algunas de estas librerías se explicarán con más detenimiento en la segunda parte del taller.
2. Para la segunda parte he hecho una selección de las librerías que a mi juicio son más prácticas e interesantes. Para cada una de ellas el esquema que seguiré es el siguiente:
   1. Expondré un problema de diseño/programación y dejaré unos minutos para que penséis una solución con vuestros conocimientos *non-boost-powered*©.
   2. Tras ese tiempo, pondremos en común las posibles soluciones, que a buen seguro serán burdas aproximaciones de lo que sería posible con Boost.
   3. Presentaré y explicaré una biblioteca de Boost con la que podríamos abordar el problema.
   4. Finalmente, resolveré el problema inicial usando Boost, y algún otro ejemplo.

# Organización del taller

Este taller se va a organizar en dos partes:

1. En la primera parte vamos a hacer un repaso por las librerías que forman parte de TR1, explicando brevemente qué hace cada una y poniendo un ejemplo. Algunas de estas librerías se explicarán con más detenimiento en la segunda parte del taller.
2. Para la segunda parte he hecho una selección de las librerías que a mi juicio son más prácticas e interesantes. Para cada una de ellas el esquema que seguiré es el siguiente:
   1. Expondré un problema de diseño/programación y dejaré unos minutos para que penséis una solución con vuestros conocimientos *non-boost-powered*©.
   2. Tras ese tiempo, pondremos en común las posibles soluciones, que a buen seguro serán burdas aproximaciones de lo que sería posible con Boost.
   3. Presentaré y explicaré una biblioteca de Boost con la que podríamos abordar el problema.
   4. Finalmente, resolveré el problema inicial usando Boost, y algún otro ejemplo.

# Organización del taller

Este taller se va a organizar en dos partes:

1. En la primera parte vamos a hacer un repaso por las librerías que forman parte de TR1, explicando brevemente qué hace cada una y poniendo un ejemplo. Algunas de estas librerías se explicarán con más detenimiento en la segunda parte del taller.
2. Para la segunda parte he hecho una selección de las librerías que a mi juicio son más prácticas e interesantes. Para cada una de ellas el esquema que seguiré es el siguiente:
   1. Expondré un problema de diseño/programación y dejaré unos minutos para que penséis una solución con vuestros conocimientos *non-boost-powered*©.
   2. Tras ese tiempo, pondremos en común las posibles soluciones, que a buen seguro serán burdas aproximaciones de lo que sería posible con Boost.
   3. Presentaré y explicaré una biblioteca de Boost con la que podríamos abordar el problema.
   4. Finalmente, resolveré el problema inicial usando Boost, y algún otro ejemplo.

# Organización del taller

Este taller se va a organizar en dos partes:

1. En la primera parte vamos a hacer un repaso por las librerías que forman parte de TR1, explicando brevemente qué hace cada una y poniendo un ejemplo. Algunas de estas librerías se explicarán con más detenimiento en la segunda parte del taller.
2. Para la segunda parte he hecho una selección de las librerías que a mi juicio son más prácticas e interesantes. Para cada una de ellas el esquema que seguiré es el siguiente:
   1. Expondré un problema de diseño/programación y dejaré unos minutos para que penséis una solución con vuestros conocimientos *non-boost-powered*©.
   2. Tras ese tiempo, pondremos en común las posibles soluciones, que a buen seguro serán burdas aproximaciones de lo que sería posible con Boost.
   3. Presentaré y explicaré una biblioteca de Boost con la que podríamos abordar el problema.
   4. Finalmente, resolveré el problema inicial usando Boost, y algún otro ejemplo.

# Organización del taller

Este taller se va a organizar en dos partes:

1. En la primera parte vamos a hacer un repaso por las librerías que forman parte de TR1, explicando brevemente qué hace cada una y poniendo un ejemplo. Algunas de estas librerías se explicarán con más detenimiento en la segunda parte del taller.
2. Para la segunda parte he hecho una selección de las librerías que a mi juicio son más prácticas e interesantes. Para cada una de ellas el esquema que seguiré es el siguiente:
   1. Expondré un problema de diseño/programación y dejaré unos minutos para que penséis una solución con vuestros conocimientos *non-boost-powered*©.
   2. Tras ese tiempo, pondremos en común las posibles soluciones, que a buen seguro serán burdas aproximaciones de lo que sería posible con Boost.
   3. Presentaré y explicaré una biblioteca de Boost con la que podríamos abordar el problema.
   4. Finalmente, resolveré el problema inicial usando Boost, y algún otro ejemplo.

# Organización del taller

Este taller se va a organizar en dos partes:

1. En la primera parte vamos a hacer un repaso por las librerías que forman parte de TR1, explicando brevemente qué hace cada una y poniendo un ejemplo. Algunas de estas librerías se explicarán con más detenimiento en la segunda parte del taller.
2. Para la segunda parte he hecho una selección de las librerías que a mi juicio son más prácticas e interesantes. Para cada una de ellas el esquema que seguiré es el siguiente:
   1. Expondré un problema de diseño/programación y dejaré unos minutos para que penséis una solución con vuestros conocimientos *non-boost-powered*©.
   2. Tras ese tiempo, pondremos en común las posibles soluciones, que a buen seguro serán burdas aproximaciones de lo que sería posible con Boost.
   3. Presentaré y explicaré una biblioteca de Boost con la que podríamos abordar el problema.
   4. Finalmente, resolveré el problema inicial usando Boost, y algún otro ejemplo.

# ¿Qué es TR1?

## ¿Qué es TR1?

C++ Technical Report 1: documento que propone una serie de adiciones a la biblioteca estándard de C++. No es un estándard oficial, sino un borrador, aunque es muy probable que forme parte del siguiente estándard `C++0x`.

Al ser una ampliación de la biblioteca estándard, no modifica aspectos base de C++ como sí hará `C++0x`, simplemente añade componentes.

# Boost.Array

`boost/array.hpp`

## boost::array

Wrapper para arrays de tamaño fijo (típicas de C), además de cumplir con las especificaciones de contenedor de la STL.

Muy parecida a `std::vector`, pero con tamaño fijo, y capacidad de ser definido como un array tradicional:

```
boost::array<string, 3> primos = { {"Shurmano", "Shurprimo", "Shurperro"} };
```

Veamos el ejemplo en `ejemplos_tr1/boost_array`

## Información y reservas

- http://www.boost.org/doc/libs/1_43_0/doc/html/array.html
- http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1548.htm

# Boost.Array

`boost/array.hpp`

## boost::array

Wrapper para arrays de tamaño fijo (típicas de C), además de cumplir con las especificaciones de contenedor de la STL.

Muy parecida a `std::vector`, pero con tamaño fijo, y capacidad de ser definido como un array tradicional:

```
boost::array<string, 3> primos = { {"Shurmano", "Shurprimo", "Shurperro"} };
```

Veamos el ejemplo en `ejemplos_tr1/boost_array`

## Información y reservas

- http://www.boost.org/doc/libs/1_43_0/doc/html/array.html
- http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1548.htm

# Boost.Array

`boost/array.hpp`

## boost::array

Wrapper para arrays de tamaño fijo (típicas de C), además de cumplir con las especificaciones de contenedor de la STL.

Muy parecida a `std::vector`, pero con tamaño fijo, y capacidad de ser definido como un array tradicional:

```
boost::array<string, 3> primos = { {"Shurmano", "Shurprimo", "Shurperro"} };
```

Veamos el ejemplo en `ejemplos_tr1/boost_array`

## Información y reservas

- `http://www.boost.org/doc/libs/1_43_0/doc/html/array.html`
- `http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1548.htm`

# Boost.Bind

*bind:* envolver, vendar, atar, amarrar, ligar.

## boost::bind

`boost::bind` es una generalización de `std::bind1st` y `std::bind2nd`. Nos permite *bindear* una serie de argumentos a cualquier cosa que *tenga pinta de función*: punteros a función, objetos función, y punteros a función-miembro.

What?

La veremos en la segunda parte del taller.

# Boost.Bind

*bind:* envolver, vendar, atar, amarrar, ligar.

**boost::bind**

`boost::bind` es una generalización de `std::bind1st` y `std::bind2nd`. Nos permite *bindear* una serie de argumentos a cualquier cosa que *tenga pinta de función*: punteros a función, objetos función, y punteros a función-miembro.

## What?

La veremos en la segunda parte del taller.

# Boost.Bind

*bind:* envolver, vendar, atar, amarrar, ligar.

## boost::bind

`boost::bind` es una generalización de `std::bind1st` y `std::bind2nd`. Nos permite *bindear* una serie de argumentos a cualquier cosa que *tenga pinta de función*: punteros a función, objetos función, y punteros a función-miembro.

## What?

La veremos en la segunda parte del taller.

# Boost.Function

## boost::function

`boost::function` nos proporciona *alojamiento* de funciones, funciones objeto y funciones miembro para su posterior ejecución. Se suele utilizar en conjunto con Boost::bind.

La veremos en la segunda parte del taller.

# Boost.Function

## boost::function

`boost::function` nos proporciona *alojamiento* de funciones, funciones objeto y funciones miembro para su posterior ejecución. Se suele utilizar en conjunto con Boost::bind.

La veremos en la segunda parte del taller.

# Boost.Hash

`boost/functional/hash.hpp`

## boost::hash

`boost::hash` nos permite generar hashes para enteros, flotantes, punteros y cadenas de forma nativa, y además provee soporte para la generación de hashes para tipos de datos definidos por el usuario y contenedores.

Veamos el ejemplo en `ejemplos_tr1/boost_hash`

# Boost.Hash

`boost/functional/hash.hpp`

## boost::hash

`boost::hash` nos permite generar hashes para enteros, flotantes, punteros y cadenas de forma nativa, y además provee soporte para la generación de hashes para tipos de datos definidos por el usuario y contenedores.

Veamos el ejemplo en `ejemplos_tr1/boost_hash`

# Boost.Mem_fn

`boost/mem_fn.hpp`

## boost::mem_fn

`boost::mem_fn` se podría definir como un caso concreto de `boost::bind`. Sirve para generar objetos función que apuntan a funciones miembro. De ahí el nombre: mem_fn = member function.

## Acerca del solapamiento con bind

`http://stackoverflow.com/questions/3088058/whats-the-point-of-using-boostmem-fn-if-we-have-boostbind`

Veamos el ejemplo en `ejemplos_tr1/boost_mem_fn`

# Boost.Mem_fn

`boost/mem_fn.hpp`

## boost::mem_fn

`boost::mem_fn` se podría definir como un caso concreto de `boost::bind`. Sirve para generar objetos función que apuntan a funciones miembro. De ahí el nombre: mem_fn = member function.

## Acerca del solapamiento con bind

`http://stackoverflow.com/questions/3088058/whats-the-point-of-using-boostmem-fn-if-we-have-boostbind`

Veamos el ejemplo en `ejemplos_tr1/boost_mem_fn`

# Boost.Mem_fn

`boost/mem_fn.hpp`

## boost::mem_fn

`boost::mem_fn` se podría definir como un caso concreto de `boost::bind`. Sirve para generar objetos función que apuntan a funciones miembro. De ahí el nombre: mem_fn = member function.

## Acerca del solapamiento con bind

`http://stackoverflow.com/questions/3088058/whats-the-point-of-using-boostmem-fn-if-we-have-boostbind`

Veamos el ejemplo en `ejemplos_tr1/boost_mem_fn`

# Boost.Random

`boost/random/*`

## boost::random

`boost::random` sirve, como su nombre indica, para trabajar con números aleatorios. Proporciona una serie de generadores aleatorios, de diferente eficiencia, y un conjunto de distribuciones a las que podemos mapear la salida de los generadores para facilitar las cosas, en lugar de andar nosotros haciendo el módulo y tal.

Veamos el ejemplo en `ejemplos_tr1/boost_random`

# Boost.Random

`boost/random/*`

## boost::random

`boost::random` sirve, como su nombre indica, para trabajar con números aleatorios. Proporciona una serie de generadores aleatorios, de diferente eficiencia, y un conjunto de distribuciones a las que podemos mapear la salida de los generadores para facilitar las cosas, en lugar de andar nosotros haciendo el módulo y tal.

Veamos el ejemplo en `ejemplos_tr1/boost_random`

# Boost.Ref

`boost/functional/ref.hpp`

## boost::ref

`boost::ref` sirve para pasar referencias a funciones que normalmente tomarían sus argumentos por valor. Así evitamos la duplicación, que en algunos casos no es posible.

## Disclaimer

No he podido dedicarle tiempo, así que no tengo mucha idea del tema.

# Boost.Regex

## boost::regex

`boost::regex` es una biblioteca de expresiones regulares que veremos en la segunda parte del taller.

# Boost Smart Pointers

## Boost Smart Pointers

Los `smart pointers` de boost son una serie de punteros inteligentes que se ocupan automáticamente de la gestión de la vida de los objetos, ahorrándonos tener que liberar la memoria al terminar de usarlos

La veremos en la segunda parte del taller.

# Boost.Tuple

`boost/tuple/*`

## boost::tuple

`boost::tuple` es una especie de versión generalizada de `std::pair`, en la que se puede almacenar un número arbitrario de valores de distinto tipo. Es especialmente útil cuando necesitamos que una función devuelva varios valores.

Veamos el ejemplo en `ejemplos_tr1/boost_tuple`

# Boost.Tuple

`boost/tuple/*`

## boost::tuple

`boost::tuple` es una especie de versión generalizada de `std::pair`, en la que se puede almacenar un número arbitrario de valores de distinto tipo. Es especialmente útil cuando necesitamos que una función devuelva varios valores.

Veamos el ejemplo en `ejemplos_tr1/boost_tuple`

# Boost.TypeTraits

`boost/type_traits.hpp`

## Background

Una `trait class` es una clase paramétrica que ofrece información sobre un tipo de datos en tiempo de compilación. (trait = rasgo)

## boost.typetraits

Se trata de una colección de clases trait que nos ayudarán a la hora de caracterizar un tipo de datos cuando trabajemos con programación genérica, pudiendo adaptar nuestros algoritmos según las propiedades del tipo de datos pero manteniendo la genericidad.

Veamos el ejemplo en `ejemplos_tr1/boost_typetraits`

# Boost.TypeTraits

`boost/type_traits.hpp`

## Background

Una `trait class` es una clase paramétrica que ofrece información sobre un tipo de datos en tiempo de compilación. (trait = rasgo)

## boost.typetraits

Se trata de una colección de clases trait que nos ayudarán a la hora de caracterizar un tipo de datos cuando trabajemos con programación genérica, pudiendo adaptar nuestros algoritmos según las propiedades del tipo de datos pero manteniendo la genericidad.

Veamos el ejemplo en `ejemplos_tr1/boost_typetraits`

# Boost.TypeTraits

`boost/type_traits.hpp`

## Background

Una `trait class` es una clase paramétrica que ofrece información sobre un tipo de datos en tiempo de compilación. (trait = rasgo)

## boost.typetraits

Se trata de una colección de clases trait que nos ayudarán a la hora de caracterizar un tipo de datos cuando trabajemos con programación genérica, pudiendo adaptar nuestros algoritmos según las propiedades del tipo de datos pero manteniendo la genericidad.

Veamos el ejemplo en `ejemplos_tr1/boost_typetraits`

# Boost.Unordered

`boost.unordered` proporciona alternativas no ordenadas a `set`, `map`, `multiset` y `multimap`, basándose en `boost::hash` para crear una tabla hash, mejorando el rendimiento en el acceso a los elementos.

## boost/unordered_set.hpp

Proporciona los tipos `boost::unordered_set` y `unordered_multiset`.

## boost/unordered_map.hpp

Proporciona los tipos `boost::unordered_map` y `unordered_multimap`.

Veamos el ejemplo en `ejemplos_tr1/boost_unordered`

# Boost.Unordered

`boost.unordered` proporciona alternativas no ordenadas a `set`, `map`, `multiset` y `multimap`, basándose en `boost::hash` para crear una tabla hash, mejorando el rendimiento en el acceso a los elementos.

## boost/unordered_set.hpp

Proporciona los tipos `boost::unordered_set` y `unordered_multiset`.

## boost/unordered_map.hpp

Proporciona los tipos `boost::unordered_map` y `unordered_multimap`.

Veamos el ejemplo en `ejemplos_tr1/boost_unordered`

# Boost.Unordered

`boost.unordered` proporciona alternativas no ordenadas a `set`, `map`, `multiset` y `multimap`, basándose en `boost::hash` para crear una tabla hash, mejorando el rendimiento en el acceso a los elementos.

## boost/unordered_set.hpp

Proporciona los tipos `boost::unordered_set` y `unordered_multiset`.

## boost/unordered_map.hpp

Proporciona los tipos `boost::unordered_map` y `unordered_multimap`.

Veamos el ejemplo en `ejemplos_tr1/boost_unordered`

# Boost.Unordered

`boost.unordered` proporciona alternativas no ordenadas a `set`, `map`, `multiset` y `multimap`, basándose en `boost::hash` para crear una tabla hash, mejorando el rendimiento en el acceso a los elementos.

## boost/unordered_set.hpp

Proporciona los tipos `boost::unordered_set` y `unordered_multiset`.

## boost/unordered_map.hpp

Proporciona los tipos `boost::unordered_map` y `unordered_multimap`.

Veamos el ejemplo en `ejemplos_tr1/boost_unordered`

# boost::format

```
boost/format.hpp
```

## boost::format

Nos permite formatear cadenas utilizando la sintaxis típica de printf.

- Utiliza un flujo interno en el que vuelca los argumentos
- Utiliza el operador % para pasar los valores.

## Sintaxis básica

```
boost::format( cadena-formato ) % arg1 % arg2 % ... % argN
```

## También como objeto

```
boost::format miFormato("... ");
miFormato % valor1 % valor2;
```

# boost::format

## Sintaxis de la cadena de formato

Compuesta de bloques que definen el formato de cada argumento:

- `%spec` Forma clásica de printf.
- `%|spec|` Como la anterior pero no es necesario indicar el tipo de datos.
- `%N%` Forma simple, sólo se indica la posición del argumento.

## Sintaxis de `spec`

`[ N$ ][ flags ][ ancho ][ .precisión ][caracter-de-tipo]`

- `[ N$ ]` Referencia al argumento N-ésimo.
- `[ flags ]` Posibles banderas de formato:
  - `'-'` alineación a la izquierda.
  - `'='` alineación centrada.
  - `'_'` alineación interna.
  - `'+'` mostrar signo incluso para números positivos.
  - `'0'` rellenar con ceros.
- `[ ancho ]` ancho mínimo. Relleno con espacios.

# boost::format

## Tabulación absoluta

- `%|nt|` siendo `n` un entero, indica que se utilizarán caracteres de relleno si es necesario para que la cadena esté a `n` caracteres del inicio.
- `%|nTX|` igual que en el caso anterior, pero X indicará el caracter de relleno.

Se escribe antes de cada cadena `spec`.

# boost::format Ejemplos

```
// argumentos por posición
// format("%|2$| %1$i \n") % "11" % "hola";

hola 11

// alineación centrada, ancho 15
// format("|%|=15||%|=15||%|=15||\n") % "Oficina" % "Software" % "Libre";

|    Oficina    |    Software   |     Libre     |

// Tabulación absoluta
// format("%2% %|10t|%|1$| \n") % "Godofredo" % "A1";
// format("%2% %|10t|%|1$| \n") % "Godofredo" % "A12345";

A1        Godofredo
A12345    Godofredo

// Notación de números
// format("%|30.5| \n") % 12312344.42345823904;
// format("%|+30.5| \n") % 12312344.42345823904;
// format("%|+30.5e| \n") % 12312344.42345823904;
// format("%|+30.5f| \n") % 123123.42345823904;

                  1.2312e+07
                 +1.2312e+07
                +1.23123e+07
               +123123.42346
```

# boost::regex

```
boost/regex.hpp
```

## Expresiones regulares

Las expresiones regulares son unas expresiones que sirven para desribir conjuntos de cadenas. Utilizan una sintaxis bastante normalizada, y su utilidad se centra en buscar y manipular textos.

## boost::regex

Ofrece soporte para búsquedas y reemplazos en textos usando expresiones regulares de manera muy sencilla e intuitiva.

## Enlazado

Es una de las pocas bibliotecas de Boost que necesita de enlazado: `-lboost_regex`

# boost::regex

```
boost/regex.hpp
```

## Expresiones regulares

Las expresiones regulares son unas expresiones que sirven para desribir conjuntos de cadenas. Utilizan una sintaxis bastante normalizada, y su utilidad se centra en buscar y manipular textos.

## boost::regex

Ofrece soporte para búsquedas y reemplazos en textos usando expresiones regulares de manera muy sencilla e intuitiva.

## Enlazado

Es una de las pocas bibliotecas de Boost que necesita de enlazado: `-lboost_regex`

# boost::regex

```
boost/regex.hpp
```

## Expresiones regulares

Las expresiones regulares son unas expresiones que sirven para desribir conjuntos de cadenas. Utilizan una sintaxis bastante normalizada, y su utilidad se centra en buscar y manipular textos.

## boost::regex

Ofrece soporte para búsquedas y reemplazos en textos usando expresiones regulares de manera muy sencilla e intuitiva.

## Enlazado

Es una de las pocas bibliotecas de Boost que necesita de enlazado: `-lboost_regex`

# Tipos de datos

## boost::regex

`boost::regex` se usa para definir la expresión regular. Viene en dos versiones: `regex` y `wregex`.

```
boost::regex patron("\\w+(\\s+\\w+){2,}\\s*");
```

## boost::match_results

`match_results` es una clase paramétrica que guarda información sobre las ocurrencias encontradas al realizar una operación con una expresión regular.

- `cmatch` para char *
- `wcmatch` para wchar *
- `smatch` para string
- `wsmatch` para wstring

# Tipos de datos

## boost::regex

`boost::regex` se usa para definir la expresión regular. Viene en dos versiones: `regex` y `wregex`.

```
boost::regex patron("\\w+(\\s+\\w+){2,}\\s*");
```

## boost::match_results

`match_results` es una clase paramétrica que guarda información sobre las ocurrencias encontradas al realizar una operación con una expresión regular.

- `cmatch` para char *
- `wcmatch` para wchar *
- `smatch` para string
- `wsmatch` para wstring

# Funciones libres

## Coincidencia total

`boost::regex_match` nos sirve para ver si una cadena concuerda **en su totalidad** con una expresión regular.

```
bool boost::regex_match(cadena, resultado, patrón);
```

## Coincidencia parcial

`boost::regex_search` se usar para buscar alguna coincidencia de la expresión regular dentro de una cadena.

```
bool boost::regex_search(cadena, resultado, patrón);
```

## Reemplazo

`boost::regex_replace` reemplaza ocurrencias de la expresión regular en la cadena con otra en el formato que indiquemos.

```
string boost::regex_replace(cadena, patron, reemplazo);
```

# Funciones libres

## Coincidencia total

`boost::regex_match` nos sirve para ver si una cadena concuerda **en su totalidad** con una expresión regular.

```
bool boost::regex_match(cadena, resultado, patrón);
```

## Coincidencia parcial

`boost::regex_search` se usar para buscar alguna coincidencia de la expresión regular dentro de una cadena.

```
bool boost::regex_search(cadena, resultado, patrón);
```

## Reemplazo

`boost::regex_replace` reemplaza ocurrencias de la expresión regular en la cadena con otra en el formato que indiquemos.

```
string boost::regex_replace(cadena, patron, reemplazo);
```

# Funciones libres

## Coincidencia total

`boost::regex_match` nos sirve para ver si una cadena concuerda **en su totalidad** con una expresión regular.

```
bool boost::regex_match(cadena, resultado, patrón);
```

## Coincidencia parcial

`boost::regex_search` se usar para buscar alguna coincidencia de la expresión regular dentro de una cadena.

```
bool boost::regex_search(cadena, resultado, patrón);
```

## Reemplazo

`boost::regex_replace` reemplaza ocurrencias de la expresión regular en la cadena con otra en el formato que indiquemos.

```
string boost::regex_replace(cadena, patron, reemplazo);
```

# Iteradores

## boost::regex_iterator

`boost::regex_iterator` nos permitará iterar sobre las ocurrencias de una expresión regular sobre una cadena.

## boost::regex_token_iterator

`boost::regex_token_iterator` nos puede servir para tokenizar una cadena, usando un patrón como divisor.

Ambos iteradores vienen en cuatro sabores, según el tipo de caracter que se use. En nuestros ejemplos usaremos `std::string`, así que habría que anteponer una *s* a los nombres de los tipos

# Iteradores

## boost::regex_iterator

`boost::regex_iterator` nos permitará iterar sobre las ocurrencias de una expresión regular sobre una cadena.

## boost::regex_token_iterator

`boost::regex_token_iterator` nos puede servir para tokenizar una cadena, usando un patrón como divisor.

Ambos iteradores vienen en cuatro sabores, según el tipo de caracter que se use. En nuestros ejemplos usaremos `std::string`, así que habría que anteponer una *s* a los nombres de los tipos

# Iteradores

## boost::regex_iterator

`boost::regex_iterator` nos permitará iterar sobre las ocurrencias de una expresión regular sobre una cadena.

## boost::regex_token_iterator

`boost::regex_token_iterator` nos puede servir para tokenizar una cadena, usando un patrón como divisor.

Ambos iteradores vienen en cuatro sabores, según el tipo de caracter que se use. En nuestros ejemplos usaremos `std::string`, así que habría que anteponer una *s* a los nombres de los tipos

# boost::program_options

`boost/program_options.hpp`

## boost::program_options

`boost::program_options` nos permite parsear, de manera sencilla, efectiva y segura, los argumentos que le pasemos a los programas al ejecutarlos en la línea de comandos, así como parsear ficheros de configuración y variables de entorno.

## Enlazado

Es otra de las pocas bibliotecas de Boost que necesita de enlazado:
`-lboost_program_options`

# boost::program_options

```
boost/program_options.hpp
```

## boost::program_options

`boost::program_options` nos permite parsear, de manera sencilla, efectiva y segura, los argumentos que le pasemos a los programas al ejecutarlos en la línea de comandos, así como parsear ficheros de configuración y variables de entorno.

## Enlazado

Es otra de las pocas bibliotecas de Boost que necesita de enlazado:
`-lboost_program_options`

# Modularización

## Modularización

El proceso de parseo tiene tres partes:

- Descripción sintáctica y semántica de las opciones y argumentos.
- Parseo de la entrada, ya sea de la línea de comandos o de otro origen.
- Almacenamiento en un mapa de variables o usando referencias.

# Definición de opciones

```
namespace po = boost::program_options;
```

## Opciones

Los objetos `po::options_description` guardan la descripción de todas las opciones. Éstas suelen tener tres partes:

- **Sintaxis:** Nombre corto y largo.
- **Semántica:** ¿Recibe valores?
- **Descripción:** Información sobre la opción.

## Opciones posicionales

Hay veces en las que incluímos argumentos que no pertenecen a ninguna opción, sino que *van por libre*. Los veremos en el segundo ejemplo.

# Definición de opciones

```
namespace po = boost::program_options;
```

## Opciones

Los objetos `po::options_description` guardan la descripción de todas las opciones. Éstas suelen tener tres partes:

- **Sintaxis:** Nombre corto y largo.
- **Semántica:** ¿Recibe valores?
- **Descripción:** Información sobre la opción.

## Opciones posicionales

Hay veces en las que incluímos argumentos que no pertenecen a ninguna opción, sino que *van por libre*. Los veremos en el segundo ejemplo.

# Parseo y almacenamiento

## Almacenamiento

Se utilizará un tipo `po::variables_map` para guardar las opciones parseadas, que es un mapa multivalor (hace uso de `boost::any`).

## Parseo

Si no tenemos grandes pretensiones, podemos utilizar la función libre `po::parse_command_line`, que nos devolverá las opciones parseadas que podremos almacenar en el mapa de variables.
Para casos más elaborados podremos instanciar un parser, aplicarle opciones y luego procesar la entrada (ejemplo 2).

## Acceso

Las opciones, una vez parseadas, se guardan en el mapa de variables, al que se puede acceder como un mapa cualquiera (casteando los valores).

# Parseo y almacenamiento

## Almacenamiento

Se utilizará un tipo `po::variables_map` para guardar las opciones parseadas, que es un mapa multivalor (hace uso de `boost::any`).

## Parseo

Si no tenemos grandes pretensiones, podemos utilizar la función libre `po::parse_command_line`, que nos devolverá las opciones parseadas que podremos almacenar en el mapa de variables.
Para casos más elaborados podremos instanciar un parser, aplicarle opciones y luego procesar la entrada (ejemplo 2).

## Acceso

Las opciones, una vez parseadas, se guardan en el mapa de variables, al que se puede acceder como un mapa cualquiera (casteando los valores).

# Parseo y almacenamiento

## Almacenamiento

Se utilizará un tipo `po::variables_map` para guardar las opciones parseadas, que es un mapa multivalor (hace uso de `boost::any`).

## Parseo

Si no tenemos grandes pretensiones, podemos utilizar la función libre `po::parse_command_line`, que nos devolverá las opciones parseadas que podremos almacenar en el mapa de variables.
Para casos más elaborados podremos instanciar un parser, aplicarle opciones y luego procesar la entrada (ejemplo 2).

## Acceso

Las opciones, una vez parseadas, se guardan en el mapa de variables, al que se puede acceder como un mapa cualquiera (casteando los valores).

# boost::foreach

```
boost/foreach.hpp
```

## boost::foreach

Esta biblioteca hace proporciona exactamente lo que su nombre indica: una forma fácil de iterar sobre un contenedor usando una estructura `foreach`, presente en infinidad de lenguajes.
Proporciona una macro `BOOST_FOREACH` que recibe dos parámetros: el objeto en el que guardar el valor, y el contenedor a recorrer:

```
std::string cadena("Hola mundo");
BOOST_FOREACH(char c, cadena) {
    std::cout << c;
}
```

## Tipos *foreacheables*

- Contenedores STL.
- Arrays.
- Cadenas terminadas en `NULL`.
- Extendible a tipos del usuario.

# boost::foreach

`boost/foreach.hpp`

## boost::foreach

Esta biblioteca hace proporciona exactamente lo que su nombre indica: una forma fácil de iterar sobre un contenedor usando una estructura `foreach`, presente en infinidad de lenguajes.
Proporciona una macro `BOOST_FOREACH` que recibe dos parámetros: el objeto en el que guardar el valor, y el contenedor a recorrer:

```
std::string cadena("Hola mundo");
BOOST_FOREACH(char c, cadena) {
    std::cout << c;
}
```

## Tipos *foreacheables*

- Contenedores STL.
- Arrays.
- Cadenas terminadas en `NULL`.
- Extendible a tipos del usuario.

## Recorrido inverso

También existe `BOOST_REVERSE_FOREACH`, para iterar del final al principio.

## Poniéndolo bonito

Para evitar tener que usar la macro con mayúsculas, los `define` recomendados por `boost` son:

```
#define foreach         BOOST_FOREACH
#define reverse_foreach BOOST_REVERSE_FOREACH
```

## Problemas conocidos

- Los tipos de datos **con comas** suelen dar problemas, ya que teóricamente la única coma que debe haber es la que separa los dos argumentos del `foreach`. Para usar tipos con comas (como `std::pair`), lo mejor es usar un typedef antes.
- `boost::foreach` está implementado usando internamente **iteradores**, por lo que la adición o eliminación de elementos al contenedor que estemos recorriendo con `foreach` da lugar a la invalidación de los iteradores previamente definidos, lo que suele conllevar un comportamiento errático en tiempo de ejecución.

## Recorrido inverso

También existe `BOOST_REVERSE_FOREACH`, para iterar del final al principio.

## Poniéndolo bonito

Para evitar tener que usar la macro con mayúsculas, los `define` recomendados por `boost` son:

```
#define foreach         BOOST_FOREACH
#define reverse_foreach BOOST_REVERSE_FOREACH
```

## Problemas conocidos

- Los tipos de datos **con comas** suelen dar problemas, ya que teóricamente la única coma que debe haber es la que separa los dos argumentos del `foreach`. Para usar tipos con comas (como `std::pair`), lo mejor es usar un typedef antes.
- `boost::foreach` está implementado usando internamente **iteradores**, por lo que la adición o eliminación de elementos al contenedor que estemos recorriendo con `foreach` da lugar a la invalidación de los iteradores previamente definidos, lo que suele conllevar un comportamiento errático en tiempo de ejecución.

## Recorrido inverso

También existe `BOOST_REVERSE_FOREACH`, para iterar del final al principio.

## Poniéndolo bonito

Para evitar tener que usar la macro con mayúsculas, los `define` recomendados por `boost` son:

```
#define foreach         BOOST_FOREACH
#define reverse_foreach BOOST_REVERSE_FOREACH
```

## Problemas conocidos

- Los tipos de datos **con comas** suelen dar problemas, ya que teóricamente la única coma que debe haber es la que separa los dos argumentos del `foreach`. Para usar tipos con comas (como `std::pair`), lo mejor es usar un typedef antes.
- `boost::foreach` está implementado usando internamente **iteradores**, por lo que la adición o eliminación de elementos al contenedor que estemos recorriendo con `foreach` da lugar a la invalidación de los iteradores previamente definidos, lo que suele conllevar un comportamiento errático en tiempo de ejecución.

# boost::lexical_cast

`boost/lexical_cast.hpp`

## Background

¿Cuántas veces hemos buscado en Google *"string to int"*? ¿Cuántas veces has mirado la sintaxis de un `stringstream`? ¿Sabes qué significa la 'a' en `atoi`?

Las conversiones léxicas son un tema peliagudo y recurrente, `boost::lexical_cast` ofrece una solución genérica que funciona, fácil de usar y recordar, y con un impacto mínimo en rendimiento y tamaño del ejecutable.

## Sintaxis

Básicamente, la sintaxis es la siguiente:

```
tipoDestino lexical_cast<tipoDestino>(tipoOrigen & valor);
```

Esto, señores, es *magia potagia*. Bueno, realmente no: Básicamente, de forma interna se vierte el valor a convertir en un flujo, que se utiliza para rellenar el tipo destino.

# boost::lexical_cast

`boost/lexical_cast.hpp`

## Background

¿Cuántas veces hemos buscado en Google *"string to int"*? ¿Cuántas veces has mirado la sintaxis de un `stringstream`? ¿Sabes qué significa la 'a' en `atoi`?

Las conversiones léxicas son un tema peliagudo y recurrente, `boost::lexical_cast` ofrece una solución genérica que funciona, fácil de usar y recordar, y con un impacto mínimo en rendimiento y tamaño del ejecutable.

## Sintaxis

Básicamente, la sintaxis es la siguiente:

```
tipoDestino lexical_cast<tipoDestino>(tipoOrigen & valor);
```

Esto, señores, es *magia potagia*. Bueno, realmente no: Básicamente, de forma interna se vierte el valor a convertir en un flujo, que se utiliza para rellenar el tipo destino.

# Ampliación y errores

Puede utilizarse con cualquier tipo de datos, siempre y cuando se cumpla lo siguiente:

- El `tipoOrigen` debe tener definido el `operador <<` para flujos.
- El `tipoDestino` debe tener definido el `operador >>` para flujos.
- El `tipoDestino` debe poderse construir por copia y tener constructor por defecto.

## Excepciones

Cuando no se puede hacer la conversión, se lanza una excepción del tipo `bad_lexical_cast : public std::bad_cast`.

## Casts entre tipos numéricos

Para convertir entre tipos numéricos, en especial tipos de diferente ancho, es mejor usar `boost::numeric_cast`.

# Ampliación y errores

Puede utilizarse con cualquier tipo de datos, siempre y cuando se cumpla lo siguiente:

- El `tipoOrigen` debe tener definido el `operador <<` para flujos.
- El `tipoDestino` debe tener definido el `operador >>` para flujos.
- El `tipoDestino` debe poderse construir por copia y tener constructor por defecto.

## Excepciones

Cuando no se puede hacer la conversión, se lanza una excepción del tipo `bad_lexical_cast : public std::bad_cast`.

## Casts entre tipos numéricos

Para convertir entre tipos numéricos, en especial tipos de diferente ancho, es mejor usar `boost::numeric_cast`.

# Ampliación y errores

Puede utilizarse con cualquier tipo de datos, siempre y cuando se cumpla lo siguiente:

- El `tipoOrigen` debe tener definido el `operador <<` para flujos.
- El `tipoDestino` debe tener definido el `operador >>` para flujos.
- El `tipoDestino` debe poderse construir por copia y tener constructor por defecto.

## Excepciones

Cuando no se puede hacer la conversión, se lanza una excepción del tipo `bad_lexical_cast : public std::bad_cast`.

## Casts entre tipos numéricos

Para convertir entre tipos numéricos, en especial tipos de diferente ancho, es mejor usar `boost::numeric_cast`.

# Introducción

## Smart pointers

Un `smart pointer` es una clase de puntero que ofrece una serie de funcionalidades extra en comparación con los punteros tradicionales. La funcionalidad más efectiva es la **gestión automática de la memoria** ocupada por el puntero.

## Smart pointers en Boost

Boost nos ofrece varios tipos de smart pointers, los más importantes son:

- Scoped pointer: Puntero no copiable.
- Shared pointer: Puntero copiable.
- Weak pointer: Observador de shared pointer.

Boost también ofrece smart pointers a arrays, pero suelen dar problemas y se recomienda usar un vector de shared pointers.

# Introducción

## Smart pointers

Un `smart pointer` es una clase de puntero que ofrece una serie de funcionalidades extra en comparación con los punteros tradicionales. La funcionalidad más efectiva es la **gestión automática de la memoria** ocupada por el puntero.

## Smart pointers en Boost

Boost nos ofrece varios tipos de smart pointers, los más importantes son:

- Scoped pointer: Puntero no copiable.
- Shared pointer: Puntero copiable.
- Weak pointer: Observador de shared pointer.

Boost también ofrece smart pointers a arrays, pero suelen dar problemas y se recomienda usar un vector de shared pointers.

# Scoped pointer

`boost/scoped_ptr.hpp`

## Scoped pointer

Un scoped pointer es una clase de smart pointer que no puede copiarse, por lo que solo puede existir un punto de acceso al objeto al que apunta.

Cuando el puntero sale del ámbito, el objeto se destruye y la memoria se libera.

## Sintaxis

```
boost::scoped_ptr<MiClase> MiPuntero (new MiClase(1));
MiPuntero.reset(new MiClase(2));
```

Se puede acceder al contenido usando el operador *, acceder a la dirección con & y acceder al puntero en bruto con el método `get()`.

# Scoped pointer

```
boost/scoped_ptr.hpp
```

## Scoped pointer

Un scoped pointer es una clase de smart pointer que no puede copiarse, por lo que solo puede existir un punto de acceso al objeto al que apunta.

Cuando el puntero sale del ámbito, el objeto se destruye y la memoria se libera.

## Sintaxis

```
boost::scoped_ptr<MiClase> MiPuntero (new MiClase(1));
MiPuntero.reset(new MiClase(2));
```

Se puede acceder al contenido usando el operador *, acceder a la dirección con & y acceder al puntero en bruto con el método `get()`.

# Shared pointer

```
boost/shared_ptr.hpp
```

## Shared pointer

Un shared pointer es un tipo de smart pointer que guarda un contador de referencias al objeto al que apunta. Cada vez que se hace una copia del puntero, se aumenta el contador. Cuando se destruye uno de los shared pointer, el contador disminuye.

Cuando el contador llega a cero, quiere decir que no hay más punteros apuntando al objeto, por lo que éste puede destruirse y liberar la memoria que ocupa. Todo esto se hace de forma transparente al usuario.

## Sintaxis

```
boost::shared_ptr<MiClase> MiPuntero (new MiClase(1));
boost::shared_ptr<MiClase> OtroPuntero = MiPuntero;
MiPuntero.reset(new MiClase(2));
```

# Shared pointer

```
boost/shared_ptr.hpp
```

## Shared pointer

Un shared pointer es un tipo de smart pointer que guarda un contador de referencias al objeto al que apunta. Cada vez que se hace una copia del puntero, se aumenta el contador. Cuando se destruye uno de los shared pointer, el contador disminuye.

Cuando el contador llega a cero, quiere decir que no hay más punteros apuntando al objeto, por lo que éste puede destruirse y liberar la memoria que ocupa. Todo esto se hace de forma transparente al usuario.

## Sintaxis

```
boost::shared_ptr<MiClase> MiPuntero (new MiClase(1));
boost::shared_ptr<MiClase> OtroPuntero = MiPuntero;
MiPuntero.reset(new MiClase(2));
```

# Funciones auxiliares para shared pointers

## make_shared - `boost/make_shared.hpp`

`make_shared` es una alternativa a usar el operador `new` a la hora de inicializar un shared pointer. Se trata de una función paramétrica que nos devuelve un shared pointer a un objeto recién creado.

```
boost::shared_ptr<string> X (new string("argumentos");
// en lugar de new, usamos:
boost::shared_ptr<string> X = boost::make_shared<string>("argumentos");
```

El uso de `make_shared` es más eficiente que `new`, ya que de una vez se hace la asignación de memoria para el propio puntero y para su contenido, en lugar de crear el contenido y luego hacer un puntero que apunte a él, como ocurre con `new`.

# Funciones auxiliares para shared pointers

## `enable_shared_from_this.hpp`

Define una clase paramétrica `enable_shared_from_this`, que se usa como clase base para permitir obtener un shared pointer al objeto actual desde una función miembro, utilizando la función `shared_from_this()`.

Es imprescindible que en algún lado del código ya exista un shared pointer apuntando al objeto.

Ver ejemplo `shared_aux.cpp`

# Definiciones

## Objeto-función

Un **objeto-función** es una instancia de una clase que tiene el operador () sobrecargado, de modo que es posible utilizar el objeto como si de una función se tratara.

La mayoría de los algoritmos de la STL son capaces de recibir objetos-función para personalizar su funcionamiento, por ejemplo:

```
struct objeto{
    void operator()(int & a){  cout << "Núm: " << a << endl; }
};

int main(){
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    std::for_each(v.begin(), v.end(), objeto());
}
```

# Definiciones

```
boost/bind.hpp
boost/function.hpp
```

## boost::function

`boost::function` es un contenedor para objetos función;

```
struct objeto{
    void operator()(int a) { cout << "Núm: " << a << endl; }
};

boost::function<void(int)>f = objeto();
```

## boost::bind

`boost::bind` es una generalización de `std::bind1st` y `std::bind2nd`. Nos permite *bindear* una serie de argumentos a cualquier cosa que *tenga pinta de función*: punteros a función, objetos función, y punteros a función-miembro y devolver un objeto función que encapsula la llamada.

Es difícil de entender, así que lo mejor es verlo con ejemplos.

# Ejemplos de boost:bind

Imaginad que tenemos una función con 6 argumentos, y el 90 % de las veces que la ejecutamos, cinco de esos argumentos son los mismos. ¿No nos interesaría tener algún mecanismo para generalizar esas llamadas? Con `bind` y `function` **¡podemos!**.

Simplemente tendríamos que llamar a `bind`, decirle qué función queremos bindear, y cómo tratar los argumentos.

- Podemos darle valores a los argumentos escribiendo directamente el valor.
- Podemos reorganizar los argumentos, por ejemplo haciendo que el tercer argumento de la función sea el primero del binding. Para ello se usan los *placeholders* _1, _2, _3...

La forma de almacenar el resultado es con un objeto `boost::function`.

# Ejemplos de boost:bind

```
float fun(int a, int b, int c, int d, int e, int f){
    return a + b + c + d + e + f;
}

function<float(int)> f = bind(fun, 10, 8, _1, 12, 43, 93);
cout << f(2) << endl; // equivale a
cout << fun(10, 8, 2, 12, 43, 93) << endl;
```

## Otros casos

- Bindear a función objeto.
- Bindear a función miembro de una clase.
- Composición de funciones.

# Introducción

```
boost/any.hpp
```

## Definición

`boost::any` es una clase que nos permite almacenar y recuperar de forma segura elementos de cualquier tipo.

Se podría equiparar a usar un `void *`, pero con los beneficios de ser *type-safe*, ya que a la hora de recuperar el valor, es necesario saber el tipo de datos que estamos recuperando.

# Sintaxis

## Sintaxis

La sintaxis de `any` es muy sencilla. Podemos asignar valores directamente:

```
boost::any cualquiera = 5;
cualquiera = "Cadena";
```

Para acceder a los valores debemos conocer su tipo:

```
boost::any valor = 5;
cout << boost::any_cast<int>(valor) << endl;
```

¡Ojo! `any_cast` lanzará una excepción si el tipo es incorrecto. En cambio, si trabajamos con punteros, no lanzará excepciones, sino que devolverá un puntero nulo.

```
boost::any num = 5;
string * a;
if (!(a = boost::any_cast<string>(&num))){
    cout << "No es una cadena" << endl;
}
```

# String Algorithms

## Algoritmos para cadenas

Boost incluye una vasta colección de algoritmos para la transformación y el manejo de cadenas. Por regla general se trata de funciones y predicados muy sencillos, que realizan búsquedas, comprobaciones y transformaciones de cadenas:

- Cambios de mayúsculas y minúsculas.
- Borrado de espacios laterales (`trailing whitespace`).
- Comparaciones y comprobaciones de inicios y finales.
- Algoritmos de búsqueda.
- Algoritmos de borrado y reemplazo.
- Splitting y joining (muy útil).

`http://www.boost.org/doc/libs/1_43_0/doc/html/string_algo/quickref.html`

```
#include <boost/algorithm/string/join.hpp>

vector<string> cadenas;
cadenas.push_back("uno");
cadenas.push_back("dos");
cout << boost::algorithm::join(cadenas, ",") << endl;
```

# Otras bibliotecas

## Bibliotecas que se han quedado fuera

Como el tiempo del taller es limitado, las siguientes bibliotecas no las podremos ver pero son interesantes/importantes.

- `boost.serialization` - Serialización
- `boost.spirit` - Generación de parsers para gramáticas regulares en forma EBNF.
- `boost.signals` - Framework para implementar el patrón observer con un modelo señales y slots similar a Qt.
- `boost.filesystem` - Lectura y escritura del sistema de ficheros y gestión de rutas. Multiplataforma.
- `boost.asio` Librería para comunicación asíncrona.
- `boost.multiindex` - Contenedor que permite el uso de varios campos como índices y más.