¿Qué hay de nuevo, viejo? - C++11 y Boost

José Tomás Tocino García

theom3ga@gmail.com

Betabeers Navidad 2013

¿Qué es C++?

Bitch, please.

¿Qué es C++11?

- *Nuevo* estándar de C++.
- Nuevos elementos sintácticos.
- Nuevas bibliotecas.
- Objetivo: *"simplificar el uso y la enseñanza de C++"*.

¿Qué es Boost?

- Bibliotecas (80+) de código abierto y revisión por pares para C++.
- Escritas por los mejores programadores C++. Revisión por pares.
- 12 de estas librerías se han integrado en C++: Array, Bind, Function, Hash, mem_fn, Random, Ref, Regex, Smart Pointers, Tuple, Type Traits y Unordered.
- Muchas son *header only*.

¿Qué es C++?

Bitch, please.

¿Qué es C++11?

- *Nuevo* estándar de C++.
- Nuevos elementos sintácticos.
- Nuevas bibliotecas.
- Objetivo: *"simplificar el uso y la enseñanza de C++"*.

¿Qué es Boost?

- Bibliotecas (80+) de código abierto y revisión por pares para C++.
- Escritas por los mejores programadores C++. Revisión por pares.
- 12 de estas librerías se han integrado en C++: Array, Bind, Function, Hash, mem_fn, Random, Ref, Regex, Smart Pointers, Tuple, Type Traits y Unordered.
- Muchas son *header only*.

¿Qué es C++?

Bitch, please.

¿Qué es C++11?

- *Nuevo* estándar de C++.
- Nuevos elementos sintácticos.
- Nuevas bibliotecas.
- Objetivo: *"simplificar el uso y la enseñanza de C++"*.

¿Qué es Boost?

- Bibliotecas (80+) de código abierto y revisión por pares para C++.
- Escritas por los mejores programadores C++. Revisión por pares.
- 12 de estas librerías se han integrado en C++: Array, Bind, Function, Hash, mem_fn, Random, Ref, Regex, Smart Pointers, Tuple, Type Traits y Unordered.
- Muchas son *header only*.

¿Cómo va a ir la charla?

Repaso de lo nuevo de C++11, comentando qué partes vienen de Boost.

Compilar

Usar la opción `-std=c++11` en GCC o Clang.

¿Cómo va a ir la charla?

Repaso de lo nuevo de C++11, comentando qué partes vienen de Boost.

Compilar

Usar la opción `-std=c++11` en GCC o Clang.

¿Qué hay de nuevo, viejo?

Deducción de tipos

Por ejemplo, tenemos un mapa:

```
std::map<string, std::vector<int> > m;
```

Pillemos un iterador...

```
std::map<string, std::vector<int> >::reverse_iterator i = m.rbegin();
```

¡NOR!

```
auto i = m.rbegin();
```

Deducción de tipos

Por ejemplo, tenemos un mapa:

```
std::map<string, std::vector<int> > m;
```

Pillemos un iterador...

```
std::map<string, std::vector<int> >::reverse_iterator i = m.rbegin();
```

¡NOR!

```
auto i = m.rbegin();
```

Deducción de tipos

Por ejemplo, tenemos un mapa:

```
std::map<string, std::vector<int> > m;
```

Pillemos un iterador...

```
std::map<string, std::vector<int> >::reverse_iterator i = m.rbegin();
```

¡NOR!

```
auto i = m.rbegin();
```

Deducción de tipos

Por ejemplo, tenemos un mapa:

```
std::map<string, std::vector<int> > m;
```

Pillemos un iterador...

```
std::map<string, std::vector<int> >::reverse_iterator i = m.rbegin();
```

¡NOR!

```
auto i = m.rbegin();
```

Inicialización uniforme

Antiguamente podías hacer esto para tipos POD:

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

Ahora es posible hacerlo con cualquier tipo de datos:

```
vector<string> v = { "Hola", "Hello" };
```

Para implementarlo en nuestras clases, definir un nuevo constructor:

```
MiClase(const std::initializer_list<int> & x) {
    // Inicializar la clase con los valores de x
}
```

Inicialización uniforme

Antiguamente podías hacer esto para tipos POD:

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

Ahora es posible hacerlo con cualquier tipo de datos:

```
vector<string> v = { "Hola", "Hello" };
```

Para implementarlo en nuestras clases, definir un nuevo constructor:

```
MiClase(const std::initializer_list<int> & x) {
    // Inicializar la clase con los valores de x
}
```

Inicialización uniforme

Antiguamente podías hacer esto para tipos POD:

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

Ahora es posible hacerlo con cualquier tipo de datos:

```
vector<string> v = { "Hola", "Hello" };
```

Para implementarlo en nuestras clases, definir un nuevo constructor:

```
MiClase(const std::initializer_list<int> & x) {
    // Inicializar la clase con los valores de x
}
```

Bucles basados en rangos

¿Iterar por un vector?

```
vector<int> v { 3, 4 };

for(vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
    cout << *i << endl;
}
```

Nope. Chuck testa.

```
for ( auto i : v ) {
    cout << i << endl;
}
```

Bucles basados en rangos

¿Iterar por un vector?

```
vector<int> v { 3, 4 };

for(vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
    cout << *i << endl;
}
```

Nope. Chuck testa.

```
for ( auto i : v ) {
    cout << i << endl;
}
```

Bucles basados en rangos - Modificación de elementos

¿Y si quiero modificar los elementos del contenedor? NO PROB.

```
for ( auto & i : v ){ // Ojo a la referencia
    i = 2 * i;
}
```

Constante para puntero nulo

En C++, `NULL` equivale a 0, lo que da lugar a ambigüedades, por ejemplo:

```
void foo(char *);
void foo(int);

foo(NULL);
```

En ese caso se llamaría a `foo(int)` en vez de `foo(char*)`.

Para evitar eso, a partir de ahora usar `nullptr`:

```
foo(nullptr);
```

Tablas hash

Por fin, contenedores basados en tablas hash, formalmente *contenedores asociativos desordenados*. Estos son:

`unordered_set`, `unordered_multiset`,
`unordered_map`, `unordered_multimap`.

Portados de `boost::unordered`.

Tiempo medio en $O(1)$

Expresiones regulares

¡Expresiones regulares! Futurista total, ni que estuviéramos en el siglo XXI.

Portadas de `boost::regex`.

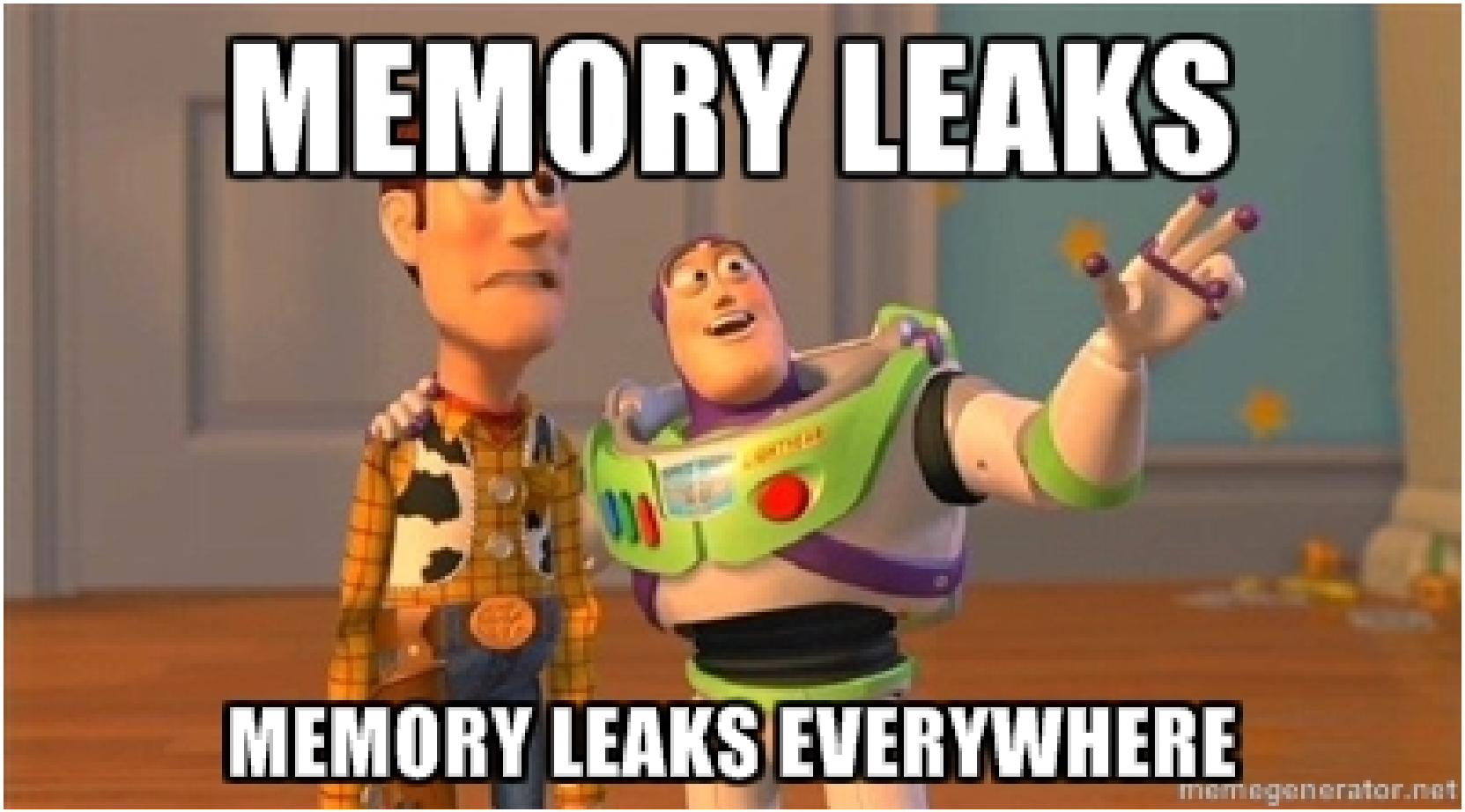
El soporte de los compiladores es *regulá*, mejor usar Boost.

Expresiones regulares, ejemplo

```
boost::regex exp(".*\\.jpe?g");

std::string ficheros[] =
  { "doc.pdf", "img1.jpg", "gurl.jpeg", "file.txt" };

for (auto & f : ficheros) {
    if (boost::regex_match(f, exp)) {
        cout << f << " es una imagen." << endl;
    } else {
        cout << f << " no es una imagen." << endl;
    }
}
```



Nunca más

Smart pointers

No tendrás que hacer `new` y `delete` nunca más gracias a los **`smart pointers`**.
Hay varios tipos:

- **`unique_ptr`**: puntero único a un recurso, nadie más puede apuntar a él. Cuando se destruye el puntero, el recurso se libera.
- **`shared_ptr`**: puntero a un recurso que puede ser compartido por más `shared_ptr`. Cuando ningún `shared_ptr` apunta al recurso, éste se libera.
- **`weak_ptr`**: puntero débil, es como un `shared_ptr` pero no se tiene en cuenta al contar referencias.

Basados en `boost::smart_pointers`.

Smart pointers, ejemplo

```
struct ClaseGordisima {
    ~ClaseGordisima() { cout << "Me destruyen!" << endl; }
};

void foo (ClaseGordisima & p) { /* ... */ }

int main()
{
    unique_ptr<ClaseGordisima> puntero { new ClaseGordisima };
    foo(*puntero);
}
// Al salir del scope se libera el recurso
```

Smart pointers, ejemplo con shared_ptr

```
struct Poseido {
    ~Poseido() { cout << "Bye :(" << endl; }
};

struct Poseedor {
    shared_ptr<Poseido> puntero;
};

int main()
{
    deque<Poseedor> p;
    p.push_back(Poseedor());
    p.push_back(Poseedor());

    p[0].puntero.reset(new Poseido);
    p[1].puntero = p[0].puntero;        // 1

    p.pop_front();      // 2
    p.pop_front();      // 3
}
```

Funciones lambda

Nueva sintaxis para definir funciones lambda:

```
auto func = [] () { cout << "Hello world"; };
func();

auto func2 = [] (int a) { return 2 * a; };
func2(4);

int something = 5;
auto func3 = [something] (int a) { return something * a; }
func3(4);
```

Ejemplo lambda

```
struct Person {
    string name;
    enum { MALE, FEMALE } gender;
};

int main(int argc, char *argv[])
{
    Person p1 { "Jose", Person::MALE };
    Person p2 { "Leti", Person::FEMALE };

    vector<Person> people { p1, p2 };

    size_t num_females = count_if(people.begin(), people.end(),
        [] (Person & p) { return p.gender == Person::FEMALE; });

    cout << "Hay " << num_females << " chicas." << endl;
}
```

Facilidades para trabajar con functors

C++11 renueva las herramientas para trabajar con funciones objeto, la mayoría copiadas de `boost` (MOAR FUNCTIONAL PLS KTHXBYE):

- `std::function`, wrapper genérico para cosas *"llamables"* (funciones, clases con `operator()`
- `std::bind`, crea wrappers para *callables* con algunos argumentos *bindeados* (toma ya). Mejor ver el ejemplo.

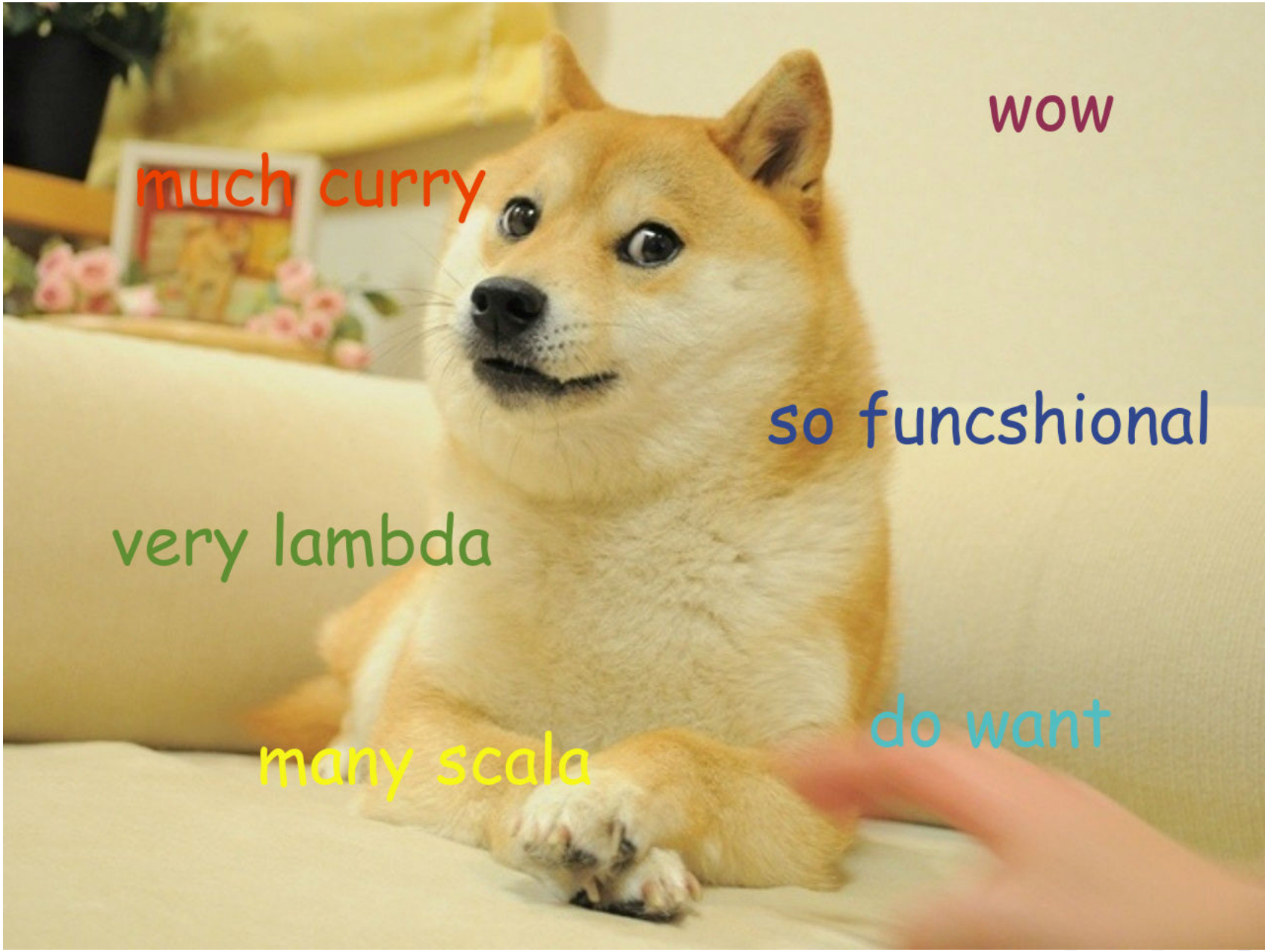
Aplicaciones parciales

Por ejemplo, aplicaciones parciales:

```
#include <functional>
using namespace std::placeholders;

float aplicarIVA (float base, float IVA) {
    return base + IVA * base;
}

int main(int argc, char *argv[])
{
    auto aplicarIVA21 = std::bind(aplicarIVA, _1, 0.21);
    cout << "5 euros + IVA = " << aplicarIVA21(5) << "euros \n";
}
```


wow
much curry
so funcshional
very lambda
do want
many scala

Soporte para threading

Ahora C++11 tiene soporte para threading de manera uniforme entre plataformas:

```
#include <thread>

void task1(string msg) {
    cout << "task1 says: " << msg;
}

int main(){
    thread t1(task1, "Hello");

    t1.join();
}
```

También viene con sus mutex, sus locks, etc. Tó la pesca.


thread t1(task);
t1.join();
cout << "wat";
throw 42;
SOY PROGRAMADOR
memegenerator.net

Promesas de futuro

Se añaden nuevas construcciones de alto nivel para tareas asíncronas, para no mancharnos las manos con hilos. Por ejemplo, `std::async`:

```
int fun(int b) { /* Calculo cosas que tardan bastante */ }

int main () {
    auto fun_futuro = std::async(fun, 20);

    // Hago otras cosas

    int resultado = fun_futuro.get();
}
```

Otras cosas de C++11

- `std::ref` (de boost)
- `std::hash` (de boost)
- `std::tuple` (de boost)
- Expresiones constantes: `constexpr`
- Referencias a rvalues (`T&&`) y semántica de movimiento.
- Delegación de constructores.
- Indicación explícita de sobrecargas y métodos finales.
- Indicación explícita de métodos borrados y predeterminados.
- Plantillas *variadic*.
- Nuevos literales de cadena con soporte para UTF (8, 16, 32).
- Literales defindos por el usuario, para hacer tus propias *"unidades"*.
- Asertos en tiempo de compilación.
- Motor de distribuciones aleatorias.

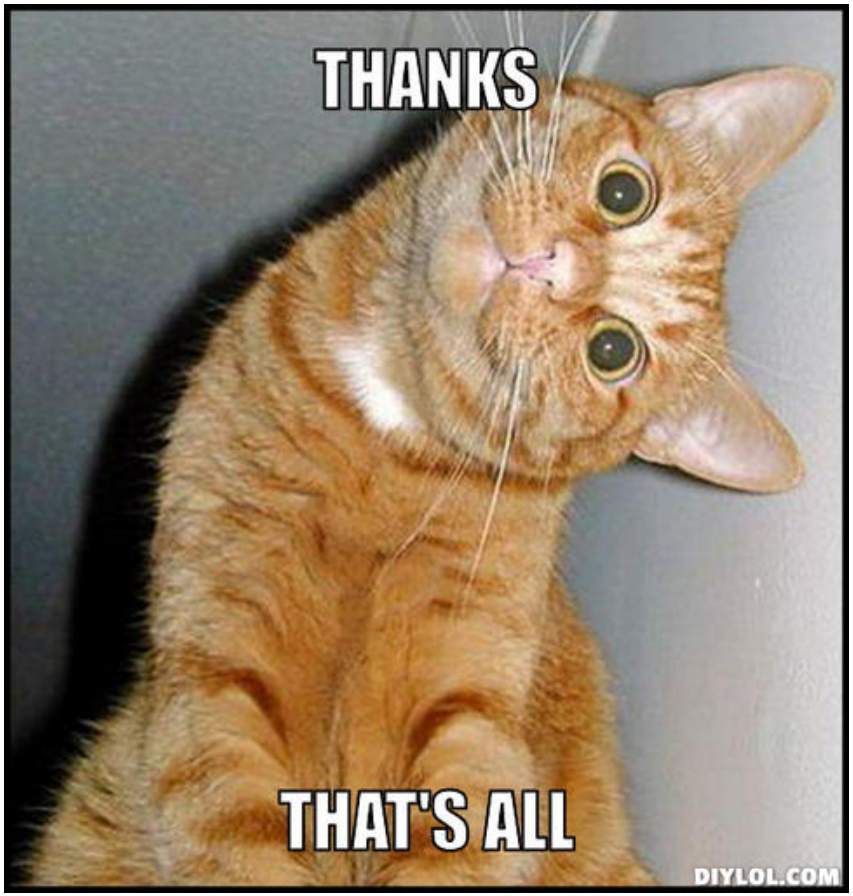
Otras cosas de Boost

- `boost::lexical_cast`
- `boost::any`
- `boost::format`
- `boost::filesystem`
- `boost::multiindex`
- `boost::signals`
- `boost::spirit` (EBNF)

Más sobre Boost

Más sobre boost en:

`https://github.com/JoseTomasTocino/boost-workshop`


THANKS
THAT'S ALL
DIYLOL.COM