www.italiancpp.org

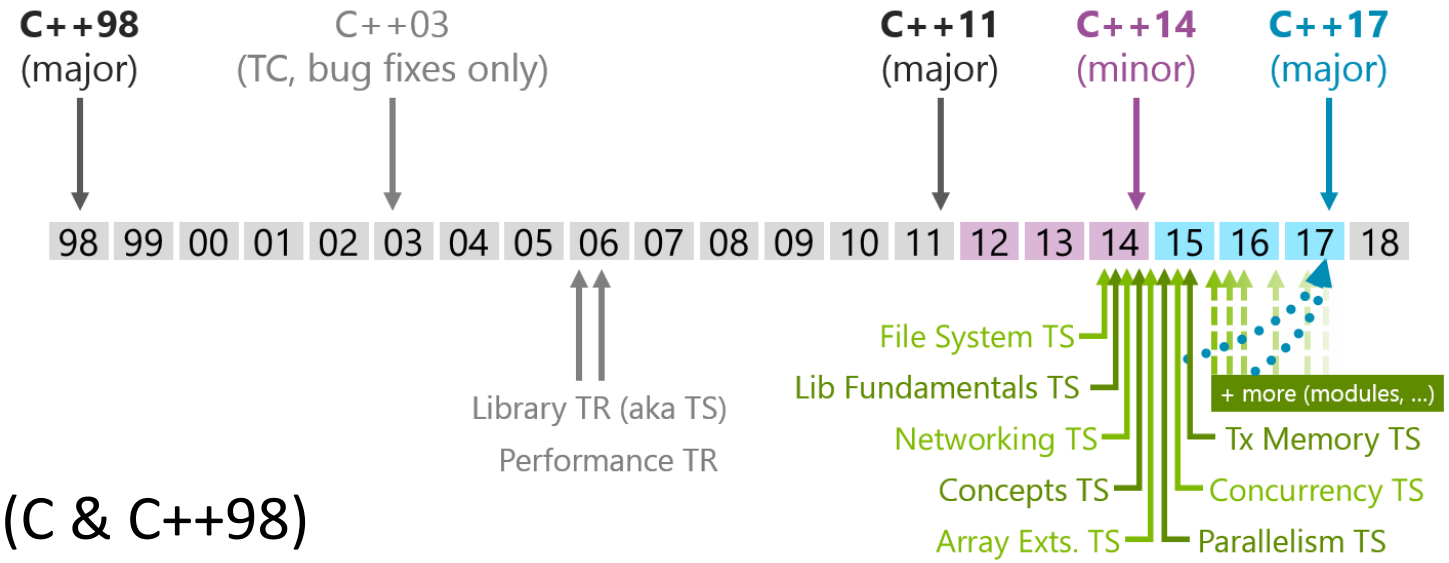
C++ and Why You Care

Gian Lorenzo Meocci

20 Giugno Firenze – C++ Community Meetup

Introduction: were we are

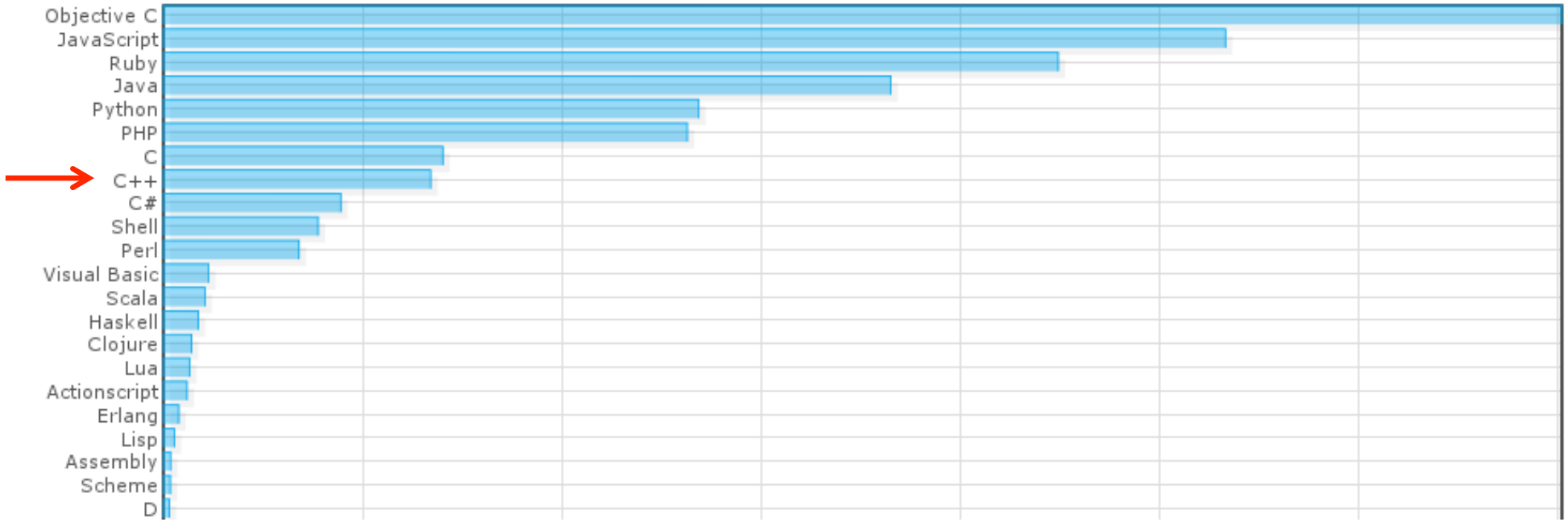
Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever.
Bjarne Stroustrup



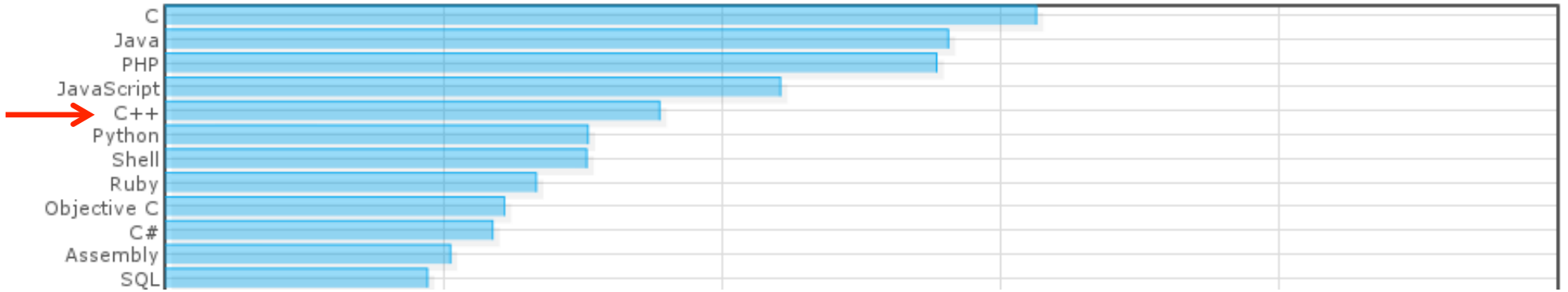
Goal of C++11/14

- Easy to teach
- Backward compatibility (C & C++98)
- Performance
- Productivity

C++ still to be wide adopted (http://langpop.com)



GitHub



Global

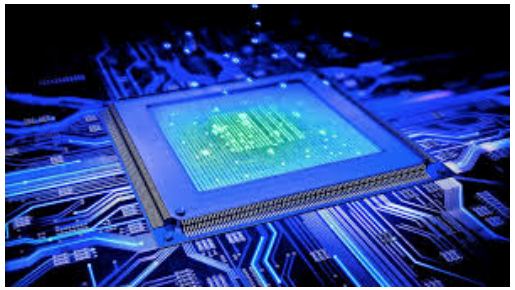
C++ in one slide (no, not really …)

```
template <typename T>
T max (T a, T b) {
    if (a > b) return a;
    else  return b;
}
```

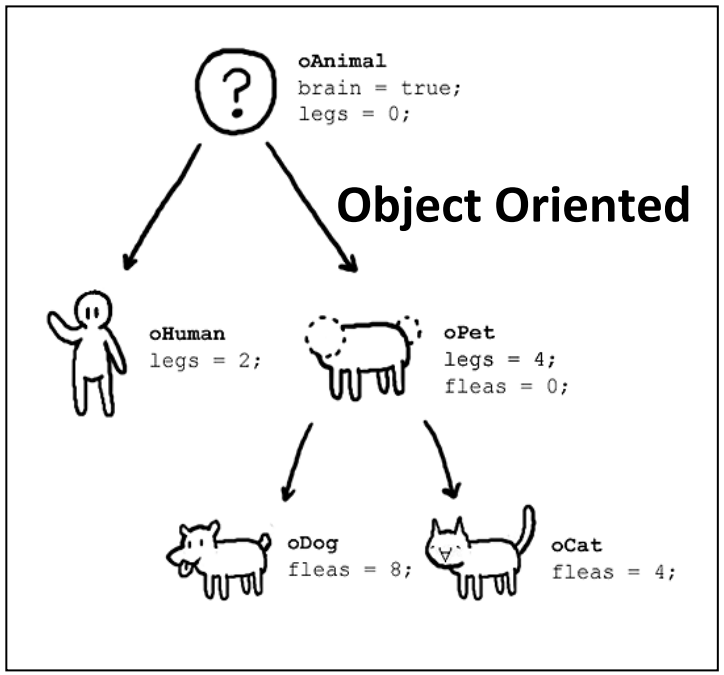
Generic

Thread Support

Strong Typed



Compiled!

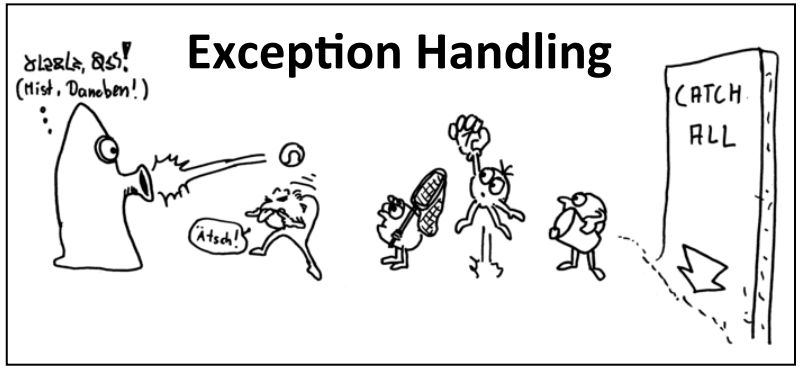


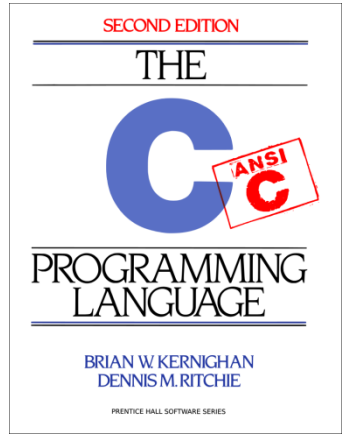
Object Oriented

Fast!!!

No GC!

Exception Handling





C super-set

C++ Fundamental strengths

Whatever we do, we should preserve C++'s fundamental strengths:

- **A direct map to hardware (initially from C)**
- **Zero-overhead abstraction (initially from Simula)**

Bjarne Stroustrup

C++ in one slide: rotate & draw

Programs must be written for people to read, and only incidentally for machines to execute.
Hal Abelson

```
void rotate_and_draw (vector<shared_ptr<Shape>>& vs, int r)
{
   // rotate all elements of vs
   for_each (vs.begin(), vs.end(), [](auto& p) { p->rotate(r); });

  // draw all elements of vs
   for (shared_ptr<Shape>& p : vs) p->draw();
}
```

- It's Object Oriented
- It's Generic
- It's Functional
- It's Modern

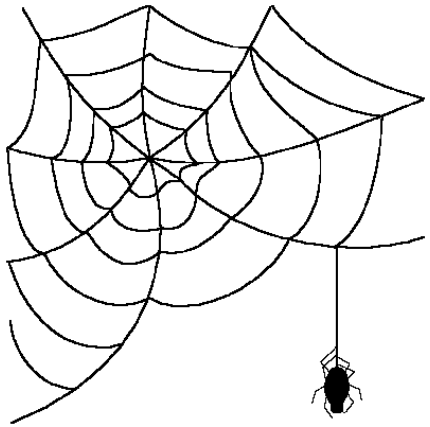
- **It's Safe?**

Myth 2: "C++ is an Object-Oriented Language". *Bjarne Stroustrup*

PERFORMANCE

Performance = Money + Battery Life + Responsive

Web



- JavaScript
- PHP
- Java
- Python
- Ruby

Cloud

- Python
- Java
- Go
- Scala
- JavaScript
- **C++**

Goal:

- Fast
- Scalable
- Cheap!

Mobile

- Java
- Objective-C
- **C++**

Goal:

- Responsive
- Power friendly
- Robust

Performance: a practical (single-thread) test

Python: simple example using generator

```
def gen_primes(n):
    if n > 0:
        yield 2
        c, e = 1, 3
        while c < n:
            if is_prime(e):
                c = c + 1
                yield e
            e = e + 2

def show_primes(n):
    for p in gen_primes(n):
        print("%d" % p)
```

Benefit:

- Reusable
- Clear
- Easy to maintain

C++14: using range-v3 (Eric Niebler)

```
auto gen_primes(uint n) {
    return view::take (
        ints(2) | filter(is_prime), n
    );
}

void show_primes(uint n)
{
    ranges::for_each(
        gen_primes(n),
        [](auto e) { cout << e << '\n';  }
    );
}
```

Performance: a practical (single-thread) test

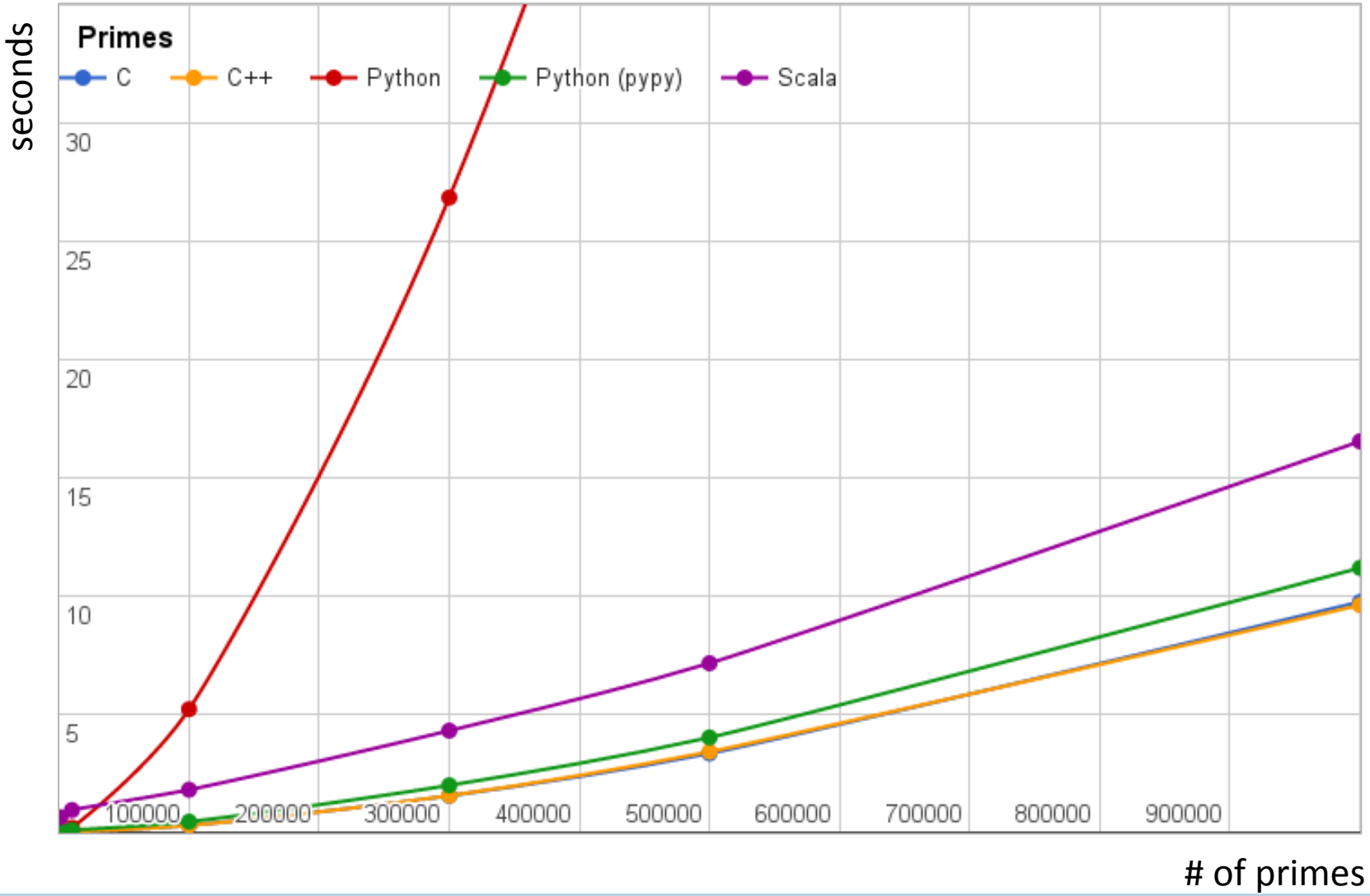
C++ allow us to write
more generic code
using High Order Function

```
template <typename F>
void primes (uint n, F f) {
   ranges::for_each(gen_primes(n), f);
}

void show_primes (uint n) {
   primes(n, [](uint e) { cout << e << '\n'; });
}

uint sum_primes (uint n) {
   uint r{};
   primes(n, [&r](uint e) { r += e; });
   return r;
}
```

Primes results

Considerations:

- **Fast as C!**
- Clear as Python
- Twice faster than Scala



RESOURCE MANAGEMENT

Herb Sutter favourites 10 lines of code

```
shared_ptr<Widget> get_widget(uint id)
{
   static map<uint, weak_ptr<Widget>> cache;
   static mutex m;

   lock_guard<mutex> locker(m);
   auto sp = cache[id].lock();
   if (!sp) cache[id] = sp = load_widget(id);
   return sp;
}
```

- **Local thread safe cache**
- **Local mutex**

- **Define a lock guard (implicit released at the end of the scope)**
- **Lookup the widget, sp is the relative shared_ptr**
- **If not exists load it & put into the cache**
- **Return it!**

C++ secret weapon: Destructor

```
class Window {
    ~Window() { cout << "~Window" << endl; }
};

struct Panel { Panel (uint id)  { … } };
class TabbedWindow : public Window
{
public:
     TabbedWindow(uint n) {
        for (uint id = 0; id < n; ++id)
            panels.push_back(make_unique<Panel>(id));
    }
     ~TabbedWindow() {cout << "~TabbedWindow" << endl; }
private:
   vector<unique_ptr<Panel>> panels;
};
```

```
cout << “Step 1" << endl;
{
     TabbedWindow mytabbed{5};
} // What happens here?
```

```
Step 1
~TabbedWindow
~Panel: p_0
~Panel: p_1
~Panel: p_2
~Window
```

C++ secret weapon: Destructor

```
class Window {
    ~Window() { cout << "~Window" << endl; }
};

struct Panel { Panel (uint id)  { … } };
class TabbedWindow : public Window
{
public:
     TabbedWindow(uint n) {
        for (uint id = 0; id < n; ++id)
            panels.push_back(make_unique<Panel>(id));
    }
     ~TabbedWindow() {cout << "~TabbedWindow" << endl; }
private:
   vector<unique_ptr<Panel>> panels;
};
```

```
cout << "Step 2" << endl;
{
   Window* w = new TabbedWindow(3);
   delete w;
}
cout << "Step 3" << endl;
{
   auto w = unique_ptr<Window>(new
TabbedWindow(3));
}
cout << "Step 4" << endl;
{
   auto w = shared_ptr<Window>(new
TabbedWindow(3));
}
```

C++ secret weapon: Destructor

```
class Window {
    ~Window() { cout << "~
};

struct Panel { Panel (uint i
class TabbedWindow : pu
{
public:
    TabbedWindow(uint n
        for (uint id = 0; id < n
            panels.push_back
    }
    ~TabbedWindow() {co
private:
   vector<unique_ptr<Par
};
```

```
cout << "Step 2" << endl;
{
   Window* w = new TabbedWindow(3);
   delete w;
}
cout << "Step 3" << endl;
{
   auto w = unique_ptr<Window>(new
TabbedWindow(3));
}
cout << "Step 4" << endl;
{
   auto w = shared_ptr<Window>(new
TabbedWindow(3));
}
```

```
Step 2                    // <---- leak here
~Window

Step 3                    // <---- leak here
~Window

Step 4
~TabbedWindow
~Panel: p_0
~Panel: p_1
~Panel: p_2
~Window
```

FUNCTIONAL

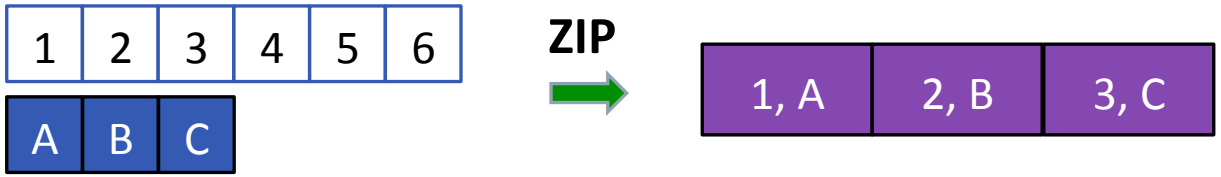
ZIP function in C++11

Let me introduce the **zip function**. This function takes two list and pairs elements from the first list with the elements from the second list.

```
zip :: [a] -> [b] -> [(a, b)]
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Haskell Implementation

Our goal is to implement it in C++11!



Thanks to lisperati.com

ZIP function in C++11

Now using C++11!

- Haskell & other ***fun.lang.*** has *tail call optimization*, in C++ is not always true so: recursive -> iterative
- If possible prefer a **std::algorithm** to raw loop

```
template <typename S, typename T, typename F>
inline void transform(const vector<S>& as, const vector<T>& bs, vector<pair<S, T>>& rs, F f)
{
    std::transform (as.cbegin(), as.cend(), bs.cbegin(), rs.begin(), f);
}

template <typename S, typename T>
vector<pair<S, T>> zip (const vector<S>& xs, const vector<T>& ys)
{
    const auto xs_size = xs.size(), ys_size = ys.size();
    vector<pair<S, T>> r(min(xs_size, ys_size));
    if (xs_size <= ys_size)
        transform (xs, ys, r, [](const S& e1, const T& e2) { return make_pair(e1, e2); });
    else
        transform (ys, xs, r, [](const T& e1, const S& e2) { return make_pair(e2, e1); });
    return r;
}
```

Interesting perspective for C++17

- **Pattern Matching**: like Haskell and Scala (already available: Mach7 lib)

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
tsum :: Num t => Tree t -> t
tsum (Leaf x) = x
tsum (Branch x y) = tsum x + tsum y

> tsum (Branch (Branch (Leaf 3) (Leaf 1)) (Leaf 7))
```

- Agumented **std::future:** .then, .next, .recover, .when_all, …
  Remember the advise of Herb: **Don't BLOCK**! Be Reactive ☺

CONCURRENCY

Threads

After several years C++ has got: *Threads*, *Memory Model* and *Async Task*.

From (pthread, boost::thread, CWinThread, Qthread, …)

to

std::thread, std::async
future/promise & Memory Model

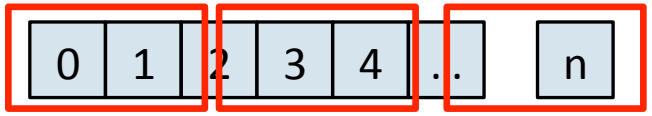
HPX (STEllAR-GROUP/hpx), **CAF**: C++ Actor Framework

Threads: an example

We want to count how many prime numbers are present in a vector. We can use this simple *serial* algorithm:

```
// load data
vector<unsigned int> v(10000);

// fill data with a sequence of positive number
generate (v.begin(), v.end(), [n=0]() mutable {return n++;});

// then compute
auto r = count_if (v.cbegin(), v.cend(), [](const auto& e){ return is_prime(e); });
cout << “Result: " << r << endl;
```

Threads: an example using async



We can simplify our code using *std::async*. This function create an “asynchronous” task and return a *future* to propagate the results of computation.

```
unsigned int count_if_async(const vector<unsigned int>& v, unsigned int n_threads)
{
    vector<future<unsigned int>> tf;

    auto body_aysnc = [&](auto it_begin, auto it_end) -> unsigned int {
        return count_if(it_begin, it_end, [](const auto& e){ return is_prime(e); });
    };

    for (auto ii : ranges(v, n_threads))
        tf.push_back(async(launch::async, body_aysnc, ii.first, ii.second));

    unsigned int r{};
    for (auto& rf : tf)
        r += rf.get();

    return r;
}
```

```
template <typename T>
using citer = typename vector<T>::const_iterator;

template <typename T>
vector<pair<citer<T>, citer<T>>> ranges(const vector<T>& v, size_t n_partions)
```

NETWORKING

Std::Net still missing into the standard

- Python has a lot of tools & framework (socket, flask, django, pyramid, …)
- Go has native support for Channel (a sort of message passing)
- Rust has std::net
- Scala & Java can use Akka, Netty, etc… (and low level socket libraries)
- Erlang / Elixir has native support for Actor Model

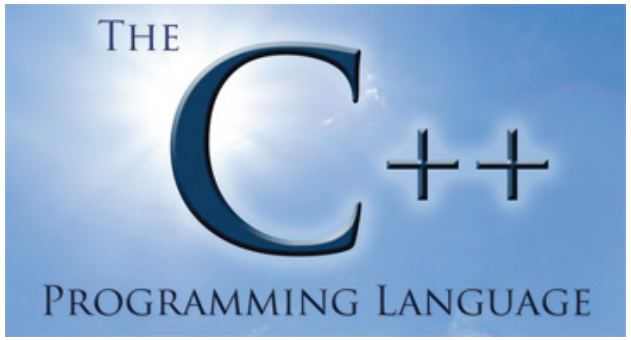
And C++ ???

- Boost Asio (probably the next std::net)
- ZeroMQ & C.
- C++ Actor Framework
- Casablanca (opensource REST framework)

What's next in C++17?

https://isocpp.org/files/papers/D4492.pdf

- Concepts (they allows us to precisely specify our generic programs and address the most vocal complaints about the quality of error messages)
- Modules (provided they can demonstrate significant isolation from macros and a significant improvement in compile times)
- Ranges and other key STL components using concepts (to improve error messages for mainstream users and improved the precision of the library specification "STL2")
- Uniform call syntax (to simplify the specification and use of template libraries)
- Co-routines (should be very fast and simple)
- Networking support (based on the asio in the TS)
- Contracts (not necessarily used in the C++17 library specification)
- SIMD vector and parallel algorithms
- Co-routines
- Library "vocabulary types", such as optional, variant, string_view, and array_view



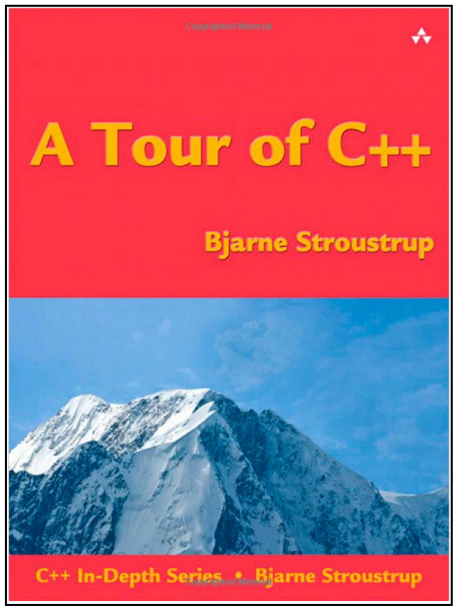
I hope to see more & more functional programming stuff in C++17 !!!

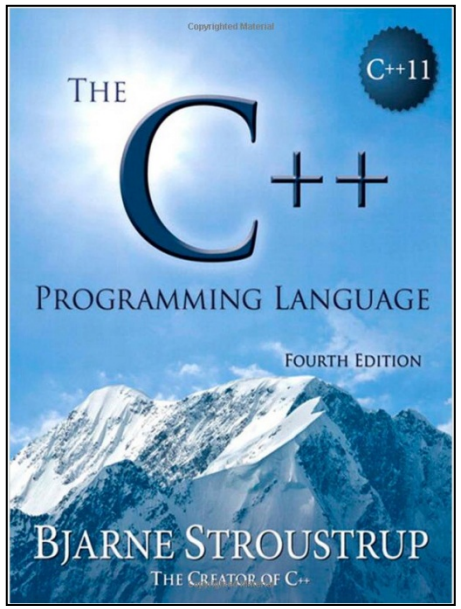
Conclusion

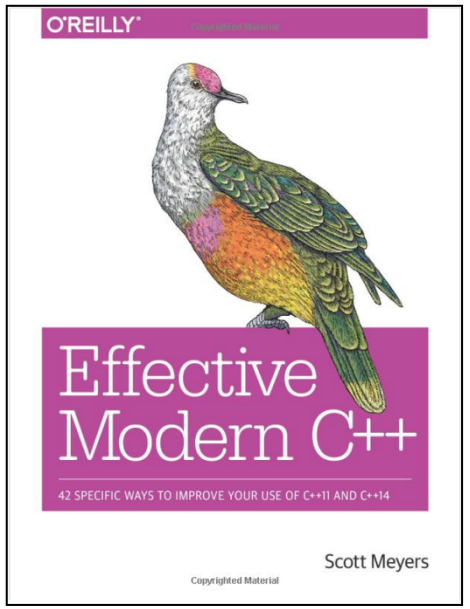
- Native ---> Fast
- Multi-Paradigm & Multi-Platform
- Runs well on Embedded & Cluster
- Good tools for control complexity
- Great library (STL, Boost, Range, Poco, HPX, CAF, …)
- Is Standard (ISO)
- Wide adopted
- A lot of resources: Books, Articles & Videos
- Many conferences (2015 Meeting C++ in Berlin)
- A great Community (like Italian C++ ☺)

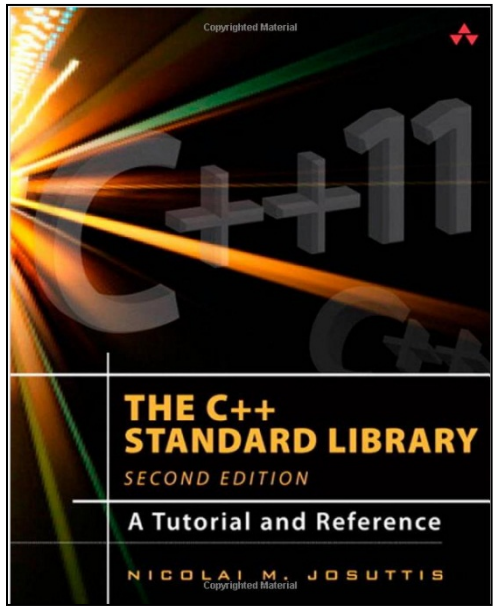
C++ Books

- Tour of C++, Bjarne Stroustrup
- The C++ Programming Language 4th edition, Bjarne Stroustrup
- Effective Modern C++, Scott Meyers
- The C++ Standard Library: A Tutorial and Reference (2nd), Nicolai M. Josuttis
- C++11 Rocks, Alex Korban









Italian C++
Community

www.italiancpp.org

Thanks!

About Me!

Software Engineer @ Commprove (Firenze)

C++ (C++11/14), JavaScript, PHP5, HTML5/CSS3, Java, Python

glmeocci@gmail.com
http://www.meocci.it
https://github.com/meox

Preferred OS: Linux, especially **Debian Linux**
Preferred PL: **C++11/14**
Preferred IDE: Sublime Text 3 & vim