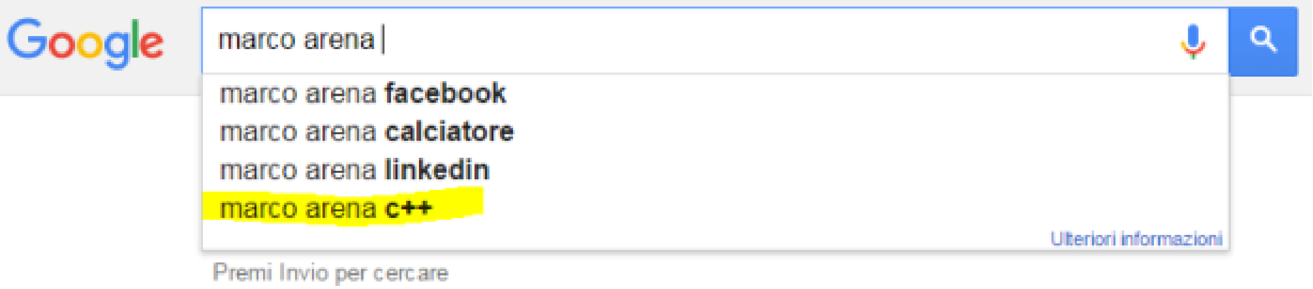
++it Italian C++ Community
www.italiancpp.org

With great C++ comes great responsibility

Marco Arena

Italian C++ Conference 2016
14 Maggio, Milano

Who I am



marco arena |

marco arena **facebook**
marco arena **calciatore**
marco arena **linkedin**
marco arena **c++**

Computer Engineer, VC++ MVP Microsoft® Most Valuable Professional

My C++ has served an Italian F1 Team since 2011

In 2013 I founded ++it, the Italian C++ Community

marco@italiancpp.org https://marcoarena.wordpress.com/

Can you help me?

Task: read an int followed by a line

```
10
this is a great event
```

```
int num; string line;
cin >> num;
getline(cin, line);
```

Task: read an int followed by a line

```
num  = 10
line = ""
```

Gadget Time!

Task: read an int followed by a line

```
10
this is a great event
```

1	0	\n	t	h	i	…

Task: read an int followed by a line

```
cin >> num >> std::ws;
getline(cin, line);
```

`getline` is an unformatted function

Is C++ hard because of such oddities?

Some programmers when they discover such oddities

C++ power & complexity

- Backwards-compatibility
- 0-overhead principle & fine-tuning control
- Independence from the paradigm & flexibility
- "Poor" standard library

Stack Overflow Programming

I'm calling in sick today because Stack Overflow is down.

Thoughts on responsibility & simplification

Thoughts on
responsibility => simplification

Understanding *Conceptual Integrity*

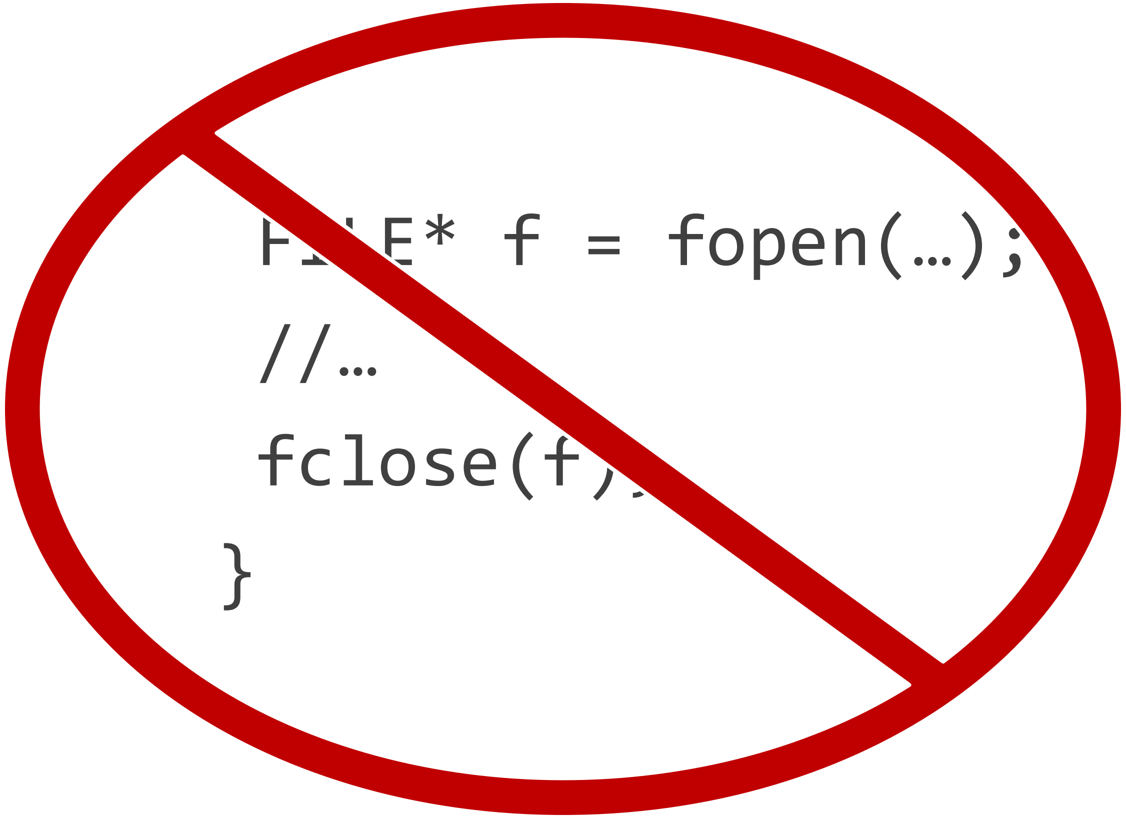
Conceptual Integrity

I will contend that **conceptual integrity** is the most important consideration in system design – it is better to have a system omit certain anomalous features and improvements, but to **reflect one set of design ideas**, than to have one that contains many good but independent and uncoordinated ideas.

[Brooks, 1975]

E.g. *On Linux, everything is a file*
On Lisp, everything is a list

RAII: Resource Acquisition is Initialization



```
  FILE* f = fopen(…);
  //…
  fclose(f);
}
```

```
  File f(…);
  //…
} // automatic fclose
```

RAII: Resource Acquisition is Initialization

RAII is possible thanks to 3 guarantees:

- Destruction **happens also in case of exceptions**
- **Order** of destruction is known (like a stack, LIFO)
- Default **destructors** are **automatically generated**

Every dynamic resource managament could (should) be done in terms of RAII

From Iterators to Ranges

```
int sum = accumulate(
                   ints(1)
                 | transform([](int i){ return i*i; })
                 | take(10)
                 , 0);
```

Task: write a stream formatting text for **OutpuDebugString**

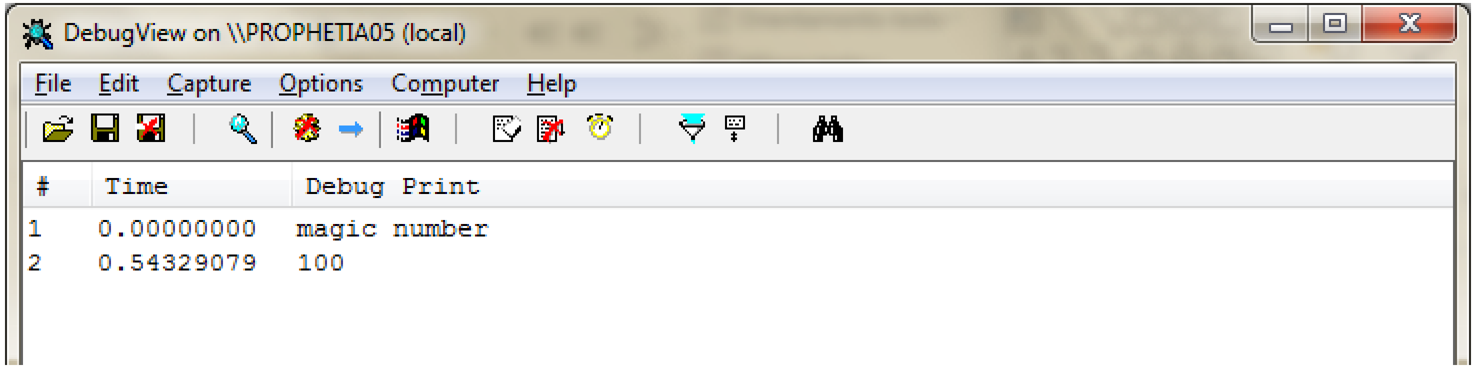
```
class debug_stream : public std::ostringstream
{
public:
    template<typename T>
    friend debug_stream& operator<<(debug_stream& os, const T& s);
};

template<typename T>
debug_stream& operator<<(debug_stream& os, const T& s)
{
    (ostringstream&) os << s;
    OutputDebugString(os.str());
    os.str(""); // clear
    return os;
}
```

What’s the problem?

```
debug_stream dbg;

dbg << "magic number " << 100 << endl;
```



DebugView on \\PROPHETIA05 (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
1   0.00000000  magic number
2   0.54329079  100
```

What is a stream?

A stream is a serial interface to any storage medium/device

Underneath the stream, a **buffer** is coupled with the device

Stream buffers decouple streams from devices

The solution: a custom stream buffer

```cpp
class dbgview_buffer : public std::stringbuf
{
public:
    int sync() override
    {
        OutputDebugString(str().c_str());
        str(""); // clear current buffer
        return 0; // ok
    }
};
```

The solution: a custom stream buffer

```
dbgview_buffer buf;
ostream dbgview(&buf);
dbgview << "Formatted string with numbers "
     << 2 << " and "
     << setprecision(3) << 10.001
     << endl; // will call «sync»
```

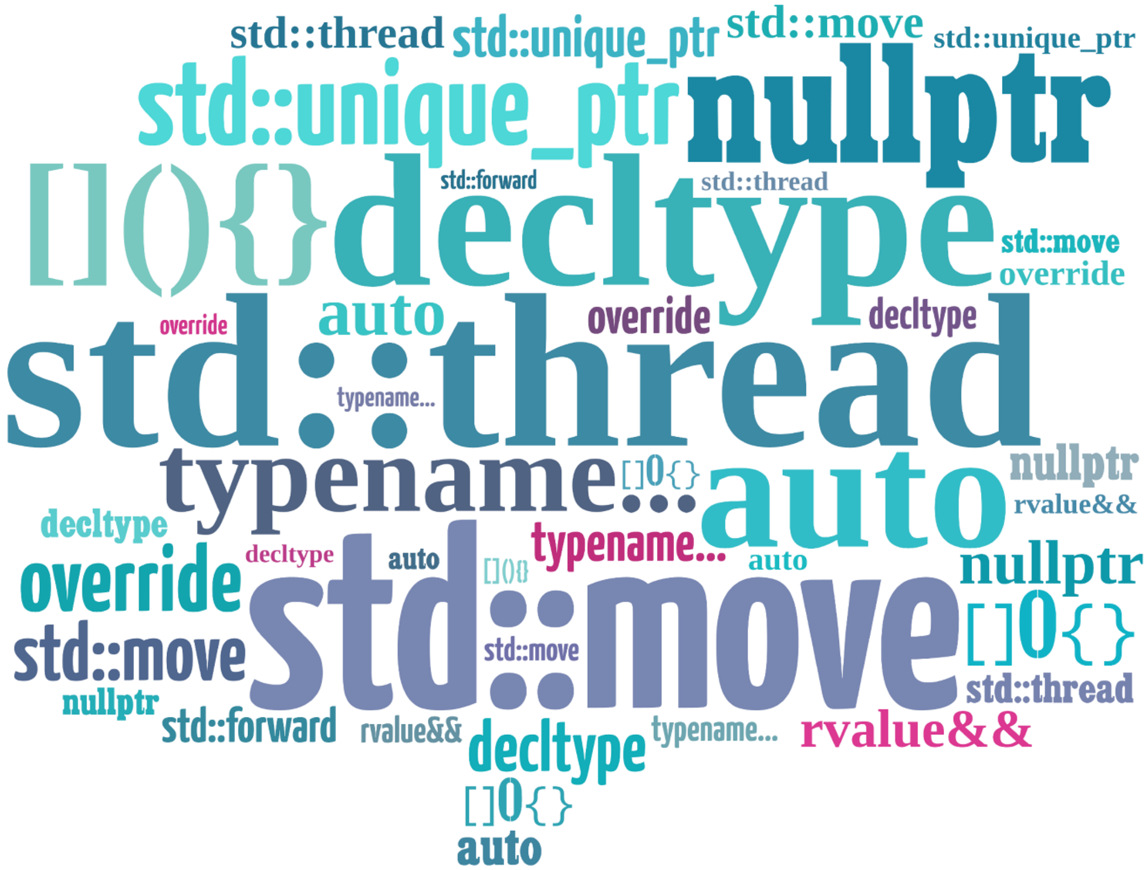
Conceptual Integrity: "Language of the Language"

Understanding **Conceptual Integrity** is mandatory not only to design **effective APIs** but also to **use the language in the proper way**.

Conceptual Integrity arises from **language constructs**
(e.g. streams and buffers, iterators),

and also from **language idioms** (e.g. RAII, move semantics).

Conceptual Integrity **evolves** along with the language (e.g. ranges).



Embracing the "new C++"

2011:
are you aware of the new C++?

2011: Start re-thinking in C++

- New features and idioms
- A few modern guidelines (Meyers, Sutter – articles/slides)
- Visual Studio 2010 already supporting TR1 and some extensions
- It has been an investment for many companies

Every new feature comes with a price

```
auto number = 10;       // auto = int
auto& ref = i;          // auto = int (auto& = int&)
auto what = ref;        // auto = int
```

Every new feature comes with a price

```
decltype(auto) look_up_a_string_1()
{
    auto str = lookup1();
    return str;
}

decltype(auto) look_up_a_string_2()
{
    auto str = lookup1();
    return (str); // ops
}
```

Every new feature comes with a price:

Learning and Awareness

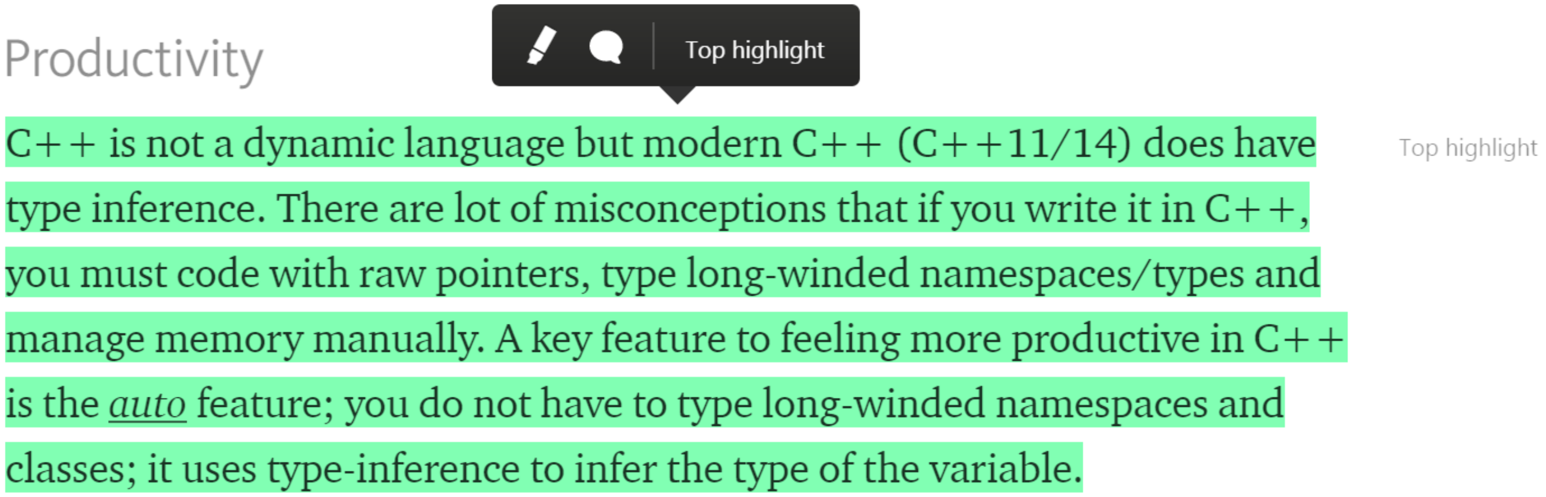
E.g.

Putting in production a new *cutting edge feature* of C++1z may be **risky** if someone of the team is not aware of that feature

Our experience since 2011/2012

- Recurring 1h/2h meetings on C++11, for some time
- Pair-programming: {fluent on C++11, less fluent on C++11}
- Setting up some team rules and doing reviews
- Some time spent on migrating (some) old code

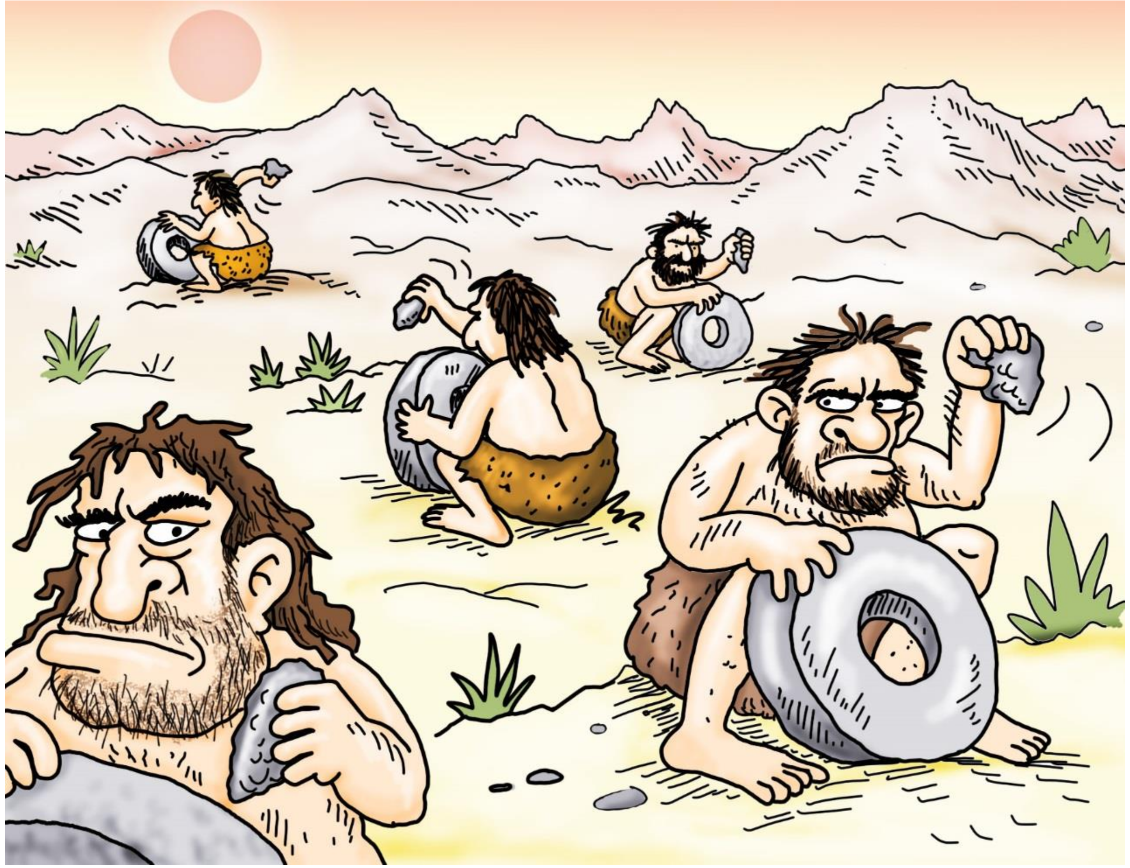
Was it worth?



Productivity

C++ is not a dynamic language but modern C++ (C++11/14) does have type inference. There are lot of misconceptions that if you write it in C++, you must code with raw pointers, type long-winded namespaces/types and manage memory manually. A key feature to feeling more productive in C++ is the *auto* feature; you do not have to type long-winded namespaces and classes; it uses type-inference to infer the type of the variable.

Starting a tech startup with C++
https://medium.com/swlh/starting-a-tech-startup-with-c-6b5d5856e6de



Don’t reinvent the wheel!

Adding enables removing

Reinventing language semantics

```
class CarSettings
{
public:
        CarSettings() : someFlag(false) {}

        CarSettings(const CarSettings& other)
                : description(other.description), someFlag(other.someFlag),


        {}

private:
        string description;
        bool someFlag;

};
```

Reinventing language semantics

```
class CarSettings
{
public:
        CarSettings() : someFlag(false) {}

        CarSettings(const CarSettings& other)
                : description(other.description), someFlag(other.someFlag),
                  coeffs(other.coeffs)

        {}

private:
        string description;
        bool someFlag;
        double coeffs[MAGIC_CONSTANT];
};
```

Reinventing language semantics

```
class CarSettings
{
public:
        CarSettings() : someFlag(false) {}

        CarSettings(const CarSettings& other)
                : description(other.description), someFlag(other.someFlag)
        {
                memcpy(coeffs, other.coeffs, sizeof(coeffs));
        }

private:
        string description;
        bool someFlag;
        double coeffs[MAGIC_CONSTANT];
};
```

Using language semantics

```
class CarSettings
{
public:
        // other functions (no special operators)
private:
        string description;
        bool someFlag = false;
        double coeffs[MAGIC_CONSTANT];
};
```

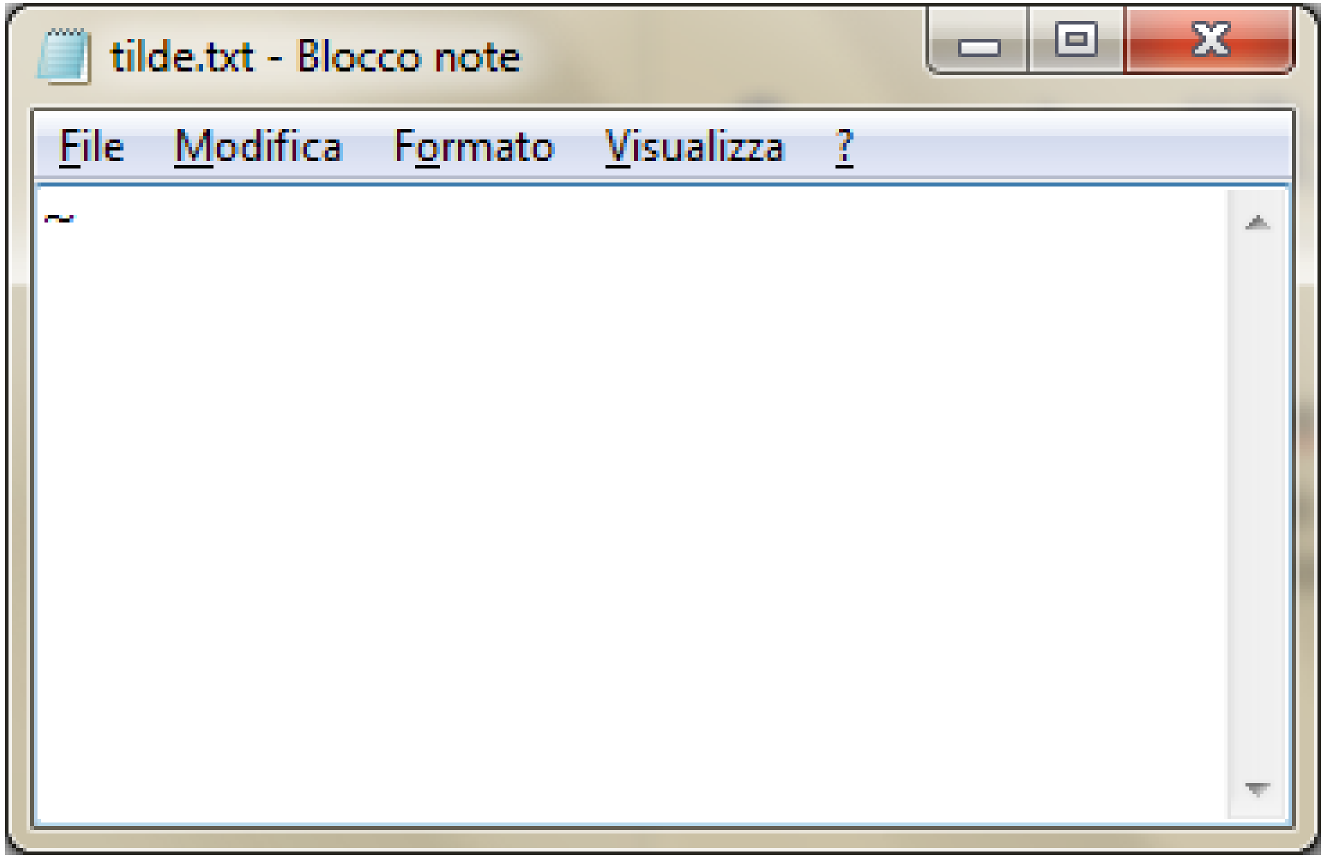
But I need special operators now…

```
class CarSettings
{
public:
        CarSettings(int N) : coeffs(new double[N]()) {}
        // dtor?
private:
        string description;
        bool someFlag = false;
        double* coeffs;
};
```

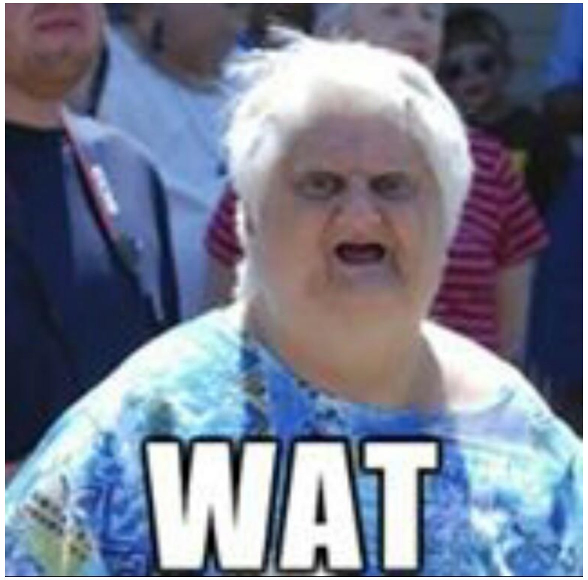
1st reason destructors are hard to write

Can’t remember how to type a tilde!

1st reason destructors are hard to write



~



Applying *Conceptual Integrity*

```
class CarSettings
{
public:
        // other functions (no special operators)
private:
        string description;
        bool someFlag = false;
        vector<double> coeffs; // or something else
};
```

(Re)Writing RAII wrappers

```
  Library lib(name);

  lib.fun1(...);

  ...

} // ~Library: Unload
```

(Re)Writing RAII wrappers

```
struct Library {
     Library(const wstring& path) : handle(LoadLibrary(path.c_str()) {}
     ~Library() { FreeLibrary(handle); }
     // what about copy/move?
     // binding functions (GetProcAddress...)
private:
     HANDLE handle;
};
```

Exploiting the STL

```
struct Library {

      Library(const wstring& path)

            : handle(LoadLibrary(path.c_str()) {}

      // clear semantics: this wrapper is movable

      // binding functions (GetProcAddress...)

private:

      unique_ptr<HANDLE, unloader> handle;

};
```

```
struct unloader{

        using pointer = HANDLE;

        void operator()(pointer h) const {

                FreeLibrary(h);

        }

};
```

The Standard Library has things you (maybe) don’t know.

Challenge:

search the STL/the language/the ecosystem

learn one new thing /investigate one aspect of C++

get results

share the **full experience** with the team and/or the **ecosystem**

Let me start

Facing *factotum* pointers

```
// what is ptr?
void f (T* ptr)
{
    ...
}
```

```
// is ptr an array?
void f (T* ptr)
{
   ptr[1];
}
```

```
// is ptr a position?
void f (T* ptr)
{
   ptr++; // next
}
```

```
// should I check ptr?
void f (T* ptr)
{
   if (ptr) {
   ...
}
```

```
// do I expect a not-null ptr?
void f (T* ptr);

if (ptr)
   f(ptr);
```

```
// should I delete ptr?
void f (T* ptr)
{
    delete ptr;
}
```

```
// will my caller delete ptr?
T* f (...)
{
    return new T(...);
}
```

```
// is ptr dangling?
void f (T* ptr)
{
   ptr->... // boom
}
```

Factotum pointers

Quick to use (when you write the code…)

Programmers intention not so clear

Comments and variable names try to replace types

Poor information for the compiler/other tools

Can we use types instead of pointers?

Let’s state a simple rule:

Use T either to indicate a position or a nullable reference*

Let’s discuss on the «nullable reference» in a few slides…

```
// gentle C-style array
void f (T* arr, int N)
{
   ...
}
```

```
void f (span<T> arr)
{
    ...
}
```

C++17

span<T> will be in C++17

A non-owning range of elements

Cheap to copy (as efficient as passing two pointers or

one pointer and an integer count)

Accessing elements is *potentially* checked

```
span<int> sp(buff, 5);
sp[10];         // potentially checked
sp[index];      // potentially checked
```

```
// I own the sequence
void f (vector<T>& arr)
{
   ...
}
```

```
// I own the sequence
void f (array<T, N>& arr)
{
   ...
}
```

```
// who owns ptr?
void f (T* ptr)
{
   ...
}
```

```
// unique ownership
void f (unique_ptr<T> obj)
{
    ...
}
```

```
// shared ownership
void f (const shared_ptr<T>&)
{
    ...
}
```

```
// modern factory
unique_ptr<T> f (...)
{
   ...
}
```

What's the matter with `nullptr`?

```
f (T* ptr) // nullptr is an option

g (T& ref) // nullptr is not an option


T someObj;

f (&someObj); // ok

g (someObj); // ok
```

What’s the matter with nullptr?

```
f (T* ptr) // nullptr is an option

g (T& ref) // nullptr shouldn’t be an option

T* ptr = nullptr;
f (ptr); // ok
g (*ptr); // UB
```

What’s the matter with **nullptr**?

```
f (T* ptr) // nullptr is an option

g (unique_ptr<T> ptr) // nullptr is an option

h (shared_ptr<T> ptr) // nullptr is an option
```

What’s the matter with **nullptr**?

```
f (?<T>)

g (?<unique_ptr<T>>)
```

nullptr will **never** be an option

What’s the matter with nullptr?

```
f (not_null<T>)

g (not_null<unique_ptr<T>>)
```

nullptr will **never** be an option

not_null<PtrType>

```
// the caller has to ensure ptr is not null
void f(not_null<int*> ptr);


// the function ensures to return not null
not_null<unique_ptr<int>> g();
```

Defeating *factotum* pointers

```
// position or nullable reference
T*

// views
T& reference_wrapper<T> not_null<T>

// range views
span<T> string_span<T> zstring
```

```
// owners
unique_ptr<T> shared_ptr<T>

vector<T> array<T, N> ...(many others)

optional<T> any<T>
```

RAII

Use T either to indicate a position or a nullable reference, use types and language constructs otherwise.*

Changing by constraining

We cannot **radically** change C++,

instead we can change **our way** to code in C++.

We cannot **radically** change C++,

instead we can constrain **our way** to code in C++.

C++ Core Guidelines

Aim to help people to use modern C++ effectively.

Rules **designed to be supported by an analysis tool**.

github.com/isocpp/CppCoreGuidelines

Ecosystem

C++ Core Guidelines – Safety Profiles

Profile: set of **deterministic** and **portably enforceable** rules that are designed to achieve a specific guarantee.

Kind of **standard** *static analysis*

Example: Lifetime Profile

http://tinyurl.com/zzvfjdb

```
void Danger(vector<int>& v)
{
        auto* p = v[0];
        if (SomeCondition)
        {
                v.push_back(23);
        }
        *p = 10; // may be "boom"...
}
```

warning C26400 Do not dereference an invalid pointer (lifetimes rule 1). 'p' was invalidated at line 8 by 'std::vector<int,std::allocator<int> >::push_back'. Path trace: 4, 6, 7, 8, 9, 11, 13, 14

For 25+ years, we have learned good things and bad things about C++.

We have understood **good constraints and idioms** for getting the best from it.

Guidelines & Safety Profiles are

standard **idioms of responsibility,**

designed to be **automatically checked.**

The future of C++ is basically:

language + library (as usual)

+ commitment to responsibility

We have a **great language.**

We have a **great responsibility.**

Thank you

Questions?