# C++11 and C++14 Outlook

**Francesco Giacomini – INFN-CNAF**

Workshop CCR

Genova, 27-31 maggio 2013

*C++11 feels like a new language – B. Stroustrup*

# A bit of history

- 1979: C++ is invented (“C with classes”)
- 1998: First ISO standard (C++98)
- 2003: “Bug fix” revision of the standard (C++03)
- 2011: Second ISO standard (C++11, aka C++0x)
- 2014 (?)
- 2017 (?)

# Why a new standard?

- Usability
  - for application developers and for library developers
- Performance
  - do more at compile time
  - do less at run time
- Bug fixing

⇒ Changes both in the core language and in the standard library

# Sources about C++11

- The standard
  - a document close to the final text is freely available at
    http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
- Official documents available at the C++ standard committee's site
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- The C++11 FAQ by B. Stroustrup
  http://www.stroustrup.com/C++11FAQ.html
- isocpp.org, stackoverflow.com, mailing lists, blogs, …

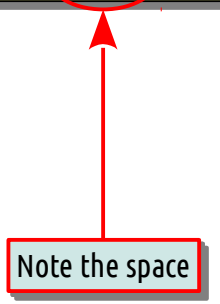
# # diff C++98 C++11

# >>

```
#include <vector>
#include <complex>

std::vector<std::complex<double>> v;  // error in C++98, ok in C++11

std::vector<std::complex<double> > v; // ok in both C++98 and C++11
```



# Initializer lists

In C++98 initializing data structures is often burdensome

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(4);
v.push_back(5);

std::map<int, std::string> ids;
ids.insert(std::make_pair(23, "Andrea"));
ids.insert(std::make_pair(49, "Camilla"));
ids.insert(std::make_pair(96, "Ugo"));
ids.insert(std::make_pair(72, "Elsa"));
```

In C++11 the operation is much simpler

```
std::vector<int> const v = {1, 2, 3, 4, 5};



std::map<int, std::string> const ids = {
  {23, "Andrea"},
  {49, "Camilla"},
  {96, "Ugo"},
  {72, "Elsa"}
};
```

Implemented with std::initializer_list<T>

# auto

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

std::map<std::string, int>::const_iterator iter = m.begin();
```

Why should I tell the compiler what the type of *iter* is? it must know!

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

auto iter = m.begin();
```

The *auto* type specifier signifies that the type of a variable being declared shall be deduced from its initializer

# auto

```
auto a;                 // error, no initializer
auto i = 0;             // i has type int
auto d = 0.;            // d has type double
auto const e = 0.;      // e has type double const
auto const& g = 0.;     // g has type double const&
auto f = 0.f;           // f has type float
auto c = "ciao";        // c has type char const*
auto p = new auto(1);   // p has type int*
auto int i;             // OK in C++98, error in C++11
```

```
template<typename T, typename U>
void op(T t, U u)
{
  …
  ??? s = t * u; // what's the type of s?
  …
}
```

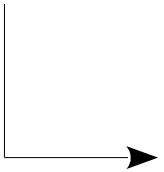
```
template<typename T, typename U>
void op(T t, U u)
{
  …
  auto s = t * u;
  …
}
```

# range for

```
std::vector<Account> v;

for (std::vector<Account>::const_iterator it = v.begin(); it != v.end(); ++it) {
  print(*it);
}

for (std::vector<Account>::iterator it = v.begin(); it != v.end(); ++it) {
  update(*it);
}
```



C++11: simplified syntax to iterate on a sequence

```
std::vector<Account> v;

for (Account a: v) { // or, rather, Account const&
  print(a);
}
for (auto u: v) { // or, rather, auto const&
  print(u);
}
for (auto& u: v) {
  update(u);
}
```

# Lambda expressions

- A concise way to create simple anonymous function objects
- Useful to pass actions/callbacks to algorithms, threads, frameworks, …

```
vector<int> v = {-2, -3, 4, 1};
sort(v.begin(), v.end()); // default sort, v == {-3, -2, 1, 4}

struct abs_compare
{
  bool operator()(int l, int r) const { return abs(l) < abs(r); }
};
sort(v.begin(), v.end(), abs_compare()); // C++98, v == {1, -2, -3, 4}

sort(v.begin(), v.end(), [](int l, int r) { return abs(l) < abs(r); }); // C++11
```

# Exception specification

- Dynamic exception specifications are deprecated
  - still supported, but can be removed from a future revision of the standard

```
void f() throw(std::runtime_exception, my_exception);
```

  - they were practically useless, if not harmful, anyway
- A *noexcept* specification has been introduced

```
void f() noexcept;
```

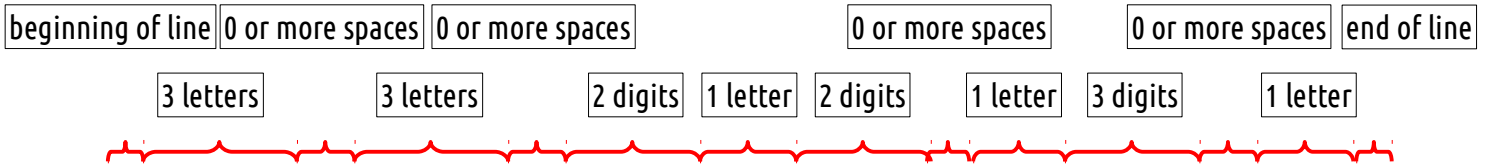
- *noexcept* tells the compiler that a function
  - doesn't throw, or
  - that the implementation is not able to manage possible exceptions, hence it is better to terminate the program if an exception occurs
- *noexcept* can depend on a constant expression, to make the function conditionally non-throwing, based on that expression

# Containers

- C++ containers belong to two broad categories
  - sequence: elements are strictly in a linear order and the order is under the control of the developer
    - vector, deque, list, **forward_list**, **array**
    - forward_list: singly-linked list
    - array<T, N>: fixed-size (N) array of elements (of type T)
  - associative: the order of the elements is under the control of the container itself, based on a key
    - set, multiset, map, multimap, **unordered_set**, **unordered_multiset**, **unordered_map**, **unordered_multimap**
    - unordered == hash

# Regular expressions

- A form of pattern-matching often used in text processing
  - think grep




```
std::regex e("^[A-Z]{3} *[A-Z]{3} *\\d{2}[A-Z]\\d{2} *[A-Z]\\d{3} *[A-Z]$");
regex_match("RSS MRA 70A01 H501 S", e); // return true
regex_match("RSS MRA 70A01 6501 S", e); // return false
```

```
std::regex e("^([A-Z]{3}) *([A-Z]{3}) *(\\d{2}[A-Z]\\d{2}) *([A-Z]\\d{3}) *([A-Z])$");
std::string fmt("\\1\\2\\3\\4\\5");
regex_replace("RSS MRA 70A01 H501 S", e, fmt); // return "RSSMRA70A01H501S"
```

# Raw string literals

- A string literal that doesn't recognize C++ escape sequences (e.g. '\n', '\\', '\"', …)
- Address readability difficulties introduced, e.g., with regular expressions and XML text

```
cout << "\\\\\"";    // print \\" (3 characters)
cout << R"(\\")";    // idem, but easier
cout << "\n";        // print a newline
cout << R"(\n)";     // print \n (2 characters)
cout << R"()")";     // error
cout << R"*+()")*+"; // print )" (2 characters)
```

an (almost) arbitrary sequence of characters

```
"^([A-Z]{3}) *([A-Z]{3}) *(\\d{2}[A-Z]\\d{2}) *([A-Z]\\d{3}) *([A-Z])$"
R"(^([A-Z]{3}) *([A-Z]{3}) *(\d{2}[A-Z]\d{2}) *([A-Z]\d{3}) *([A-Z])$)"

"\\1\\2\\3\\4\\5"
R"(\1\2\3\4\5)"
```

# Random number generation

- Generators/engines
  - generate uniformly distributed unsigned integers according to specific properties
  - linear congruential, mersenne twister, …
- Distributions
  - generate values that are distributed according to an associated mathematical probability density function or according to an associated discrete probability function
  - uniform int, uniform real, bernoulli, binomial, normal, …

```
default_random_engine e;                // a typedef for one of the engines
uniform_int_distribution<> d(1, 6);
d(e);                                   // return an integer between 1 and 6
```

# Smart pointers

- unique_ptr<T>
  - strict ownership
    - owns the object it points to and deletes it when it goes out of scope
  - movable, not copiable
  - replaces auto_ptr
  - support a custom deleter

```
unique_ptr<C> p(new C(…));
C c = *p; // use like a pointer
p->f();   // use like a pointer
v.push_back(std::move(p));
*p; // error
// do not delete
```

- shared_ptr<T>
  - shared ownership
    - the last that goes out of scope, deletes the object
  - movable and copiable
  - reference counted
  - support a custom deleter

```
shared_ptr<C> p = make_shared<C>(…);
C c = *p; // use like a pointer
p->f();   // use like a pointer
v.push_back(p);
*p; // ok
// do not delete
```

# Memory model

- Finally C++ has a memory model that contemplates a multi-threaded execution of a program
- A thread is a single flow of control within a program
  - Every thread can potentially access every object and function in the program
  - The interleaving of each thread's instructions is undefined
- C++ guarantees that two threads can update and access **separate** memory locations without interfering with each other
- For all other situations updates and accesses have to be properly synchronized
  - atomics, locks, memory fences
  - to avoid data races $\Longrightarrow$ undefined behaviour

# Moreover...

- nullptr
  - a null pointer literal
- in-class member initializers
  - initialize members directly in the class definition
- delegating constructors
  - implement a constructor in terms of another
- inheriting constructors
  - inherit constructors from base classes
- deleted functions
  - prevent the compiler from considering or generating certain functions
- defaulted functions
  - tell the compiler that the default implementation of certain special functions is ok

# Moreover...

- long long
  - an integer at least 64 bits long
- enum class
  - strongly typed enum, with better control on underlying representation and conversion to/from int
- type aliases
  - enhanced typedef
- static_assert
  - fail during compilation if a given constant expression is false
- variadic templates
  - allow the implementation of, e.g., type-safe printf and tuples
- inline namespaces
  - support for library versioning

# Moreover...

- override control
  - state explicitly that a function **override**s another in a base class or that a function is **final** (i.e. cannot be overridden)
- extern templates
  - control where a template is instantiated
- attributes
  - a mechanism to annotate source code
- Plain Old Data (POD) objects and unions
  - relaxed constraints
- user-defined literals
  - literals of user-defined types (e.g. 3s, meaning 3 seconds, in a type-safe manner)
- decltype
  - determine the type of an expression

# Moreover...

- rvalue references and move semantics
  - to optimize away temporary objects
- constexpr
  - more computation at compile time
- system error support
  - errno in C++
- exception handling
  - exception_ptr, nested_exception
- memory
  - alignment and allocators
- metaprogramming and type traits
  - type inspection and manipulation at compile time
- compile time rational arithmetic
  - representation as (num, den) and corresponding arithmetic

# Moreover...

- time utilities
  - time point, duration, clocks
- tuples
  - a heterogeneous, fixed-size collection of values
- bind and function
  - to manage callable entities
- concurrency
  - locks, threads, async, ...
- new algorithms
- …

# More Information

- On C++98
  - https://www.cnaf.infn.it/~giaco/c++/introduction_to_c++.pdf
- On C++11
  - https://www.cnaf.infn.it/~giaco/c++/c++11_padova_20130327.pdf

# C++14 Outlook

# Why C++14

*My definition of the aim of C++14 is "to complete C++11." That is, to add things we simply didn't have time to do before shipping C++11 [...]. It is not the job of C++14 to add major new features or support significant new programming techniques. That's for C++17. C++14 is a "minor release."*

*Bjarne Stroustrup*

- Committee draft for C++14 is available
  - http://isocpp.org/files/papers/N3690.pdf

# What C++ will offer in 2014

- make_unique
  - to create unique_ptr<>'s → no need to use new and delete any more
- generic lambda
  - auto as a type name
- variable template
  - e.g. to express a constant value at different precisions
- dynamic array
  - array with fixed but runtime-defined size
- optional<T>
  - to express an optional value

# What C++ will offer in 2014 /2

- Additional specifications to appear about the same time of C++14
  - **File system** library
  - **Networking** library
  - **Template constraints** (aka “concepts lite”)
    - to express constraints on template parameters → better error messages
    - language extension