VERSO C++11/14

TANTE NOVITÀ

Sum of all things C++11

- • __cplusplus
- • alignments
- • attributes
- • atomic operations
- • auto (type deduction from initializer)
- • C99 features
- • enum class (scoped and strongly typed enums)
- • copying and rethrowing exceptions
- • constant expressions (generalized and guaranteed; constexpr)
- • decltype
- • default template parameters for function
- • defaulted and deleted functions (control of defaults)
- • delegating constructors
- • Dynamic Initialization and Destruction with Concurrency
- • explicit conversion operators
- • extended friend syntax
- • extended integer types
- • extern templates
- • for statement ; see range for statement
- • generalized SFINAE rules
- • Uniform initialization syntax and semantics
- • unions (generalized)
- • user-defined literals
- • variadic templates
- • in-class member initializers
- • inherited constructors
- • initializer lists (uniform and general initialization)
- • lambdas
- • local classes as template arguments
- • long long integers (at least 64 bits)
- • memory model
- • move semantics; see rvalue references
- • Namespace Associations (Strong using)
- • Preventing narrowing
- • null pointer (nullptr)
- • PODs (generalized)
- • range for statement
- • raw string literals
- • right-angle brackets
- • rvalue references
- • static (compile-time) assertions (static_assert)
- • suffix return type syntax (extended function declaration syntax)
- • template alias
- • template typedef ; see template alias
- • thread-local storage (thread_local)
- • unicode characters

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (std::vector<int>::const_iterator it = log.cbegin();
             it != log.cend(); ++it)
         transmit_item(*it);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (std::vector<int>::const_iterator it = log.cbegin();
             it != log.cend(); ++it)
         transmit_item(*it);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-

```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (decltype(log.begin()) it = log.cbegin();
             it != log.cend(); ++it)
         transmit_item(*it);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- decltype( ) restituisce il tipo dell'espressione in argomento.
- Ma ancora meglio, conviene utilizzare la keyword `auto`

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-

```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (decltype(log.begin()) it = log.cbegin();
             it != log.cend(); ++it)
         transmit_item(*it);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-

```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (auto it = log.cbegin();
             it != log.cend(); ++it)
         transmit_item(*it);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (auto it = log.cbegin(); it != log.cend(); ++it)
         transmit_item(*it);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- **Automatic type deduction**
- La keyword auto può essere utilizzata come indicatore di tipo nella dichiarazione di una variabile
- Il compilatore deduce il tipo di variabile dichiarata esaminandone l'utilizzo e/o la definizione.

- Usare quando sia possibile dedurre il tipo, senza ambiguità
- Non abusarne: `auto main( auto argc, const auto** argv ) ....`

- C++14: auto deduce anche il return type delle funzioni
- ```
  auto add( int x, int y ) {
      return x + y;
  }
  ```
  ```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
- ```
  #include <iostream>
  #include <vector>

  static void transmit_item(int i) {
       std::cout << i << std::endl;
  }

  static void transmit_log(const std::vector<int> & log) {
       for (auto it = log.cbegin(); it != log.cend(); ++it)
           transmit_item(*it);
  }

  int main() {
       std::vector<int> log{20,24,37,42,23,45,37};
       transmit_log(log);
  }
  ```
  ```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (auto i : log)
         transmit_item(i);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- **Range-based for loop**
- Sintassi comoda per iterare su un range o sequenza di elementi
- In particolare, funziona su tutti i container della STL
- Es:
- ```
  std::vector<int> v = {3, 5, 23, -12, 0, 4};
  for( int n : v )
      std::cout << n << " ";
  ```
- Per container associativi (ad es. `std::map<>`), variabile di iterazione è key
  ```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (auto i : log)
         transmit_item(i);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (auto i : log)
         transmit_item(i);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <alghoritm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     for (auto i : log)
         transmit_item(i);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     std::for_each(log.begin(), log.end(), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- **Apply function to a range**
- Esegue – anzi, applica - una funzione su tutti gli elementi di un range
- Equivalente alla funzione `map()` di Python
- Possibile implementazione equivalente:

```
Function for_each( InputIterator first, InputIterator last, Function fn) {
     while( first != last) {
         fn( *first );
         ++first;
     }
     return fn;
}
```

- La funzione fn è una funzione unaria (viene passata come puntatore a funzione) oppure può essere un function object. Il return value della funzione viene ignorato.

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     std::for_each(log.begin(), log.end(), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
- ```
  #include <iostream>
  #include <vector>
  #include <algorithm>

  static void transmit_item(int i) {
       std::cout << i << std::endl;
  }

  static void transmit_log(const std::vector<int> & log) {
       std::for_each(std::begin(log), std::end(log), trasmit_item);
  }

  int main() {
       std::vector<int> log{20,24,37,42,23,45,37};
       transmit_log(log);
  }
  ```
  ```

C++ 11 / 14

- Supponiamo di voler ordinare il vettore prima di trasmetterlo

- Per prima cosa passo una copia locale alla funzione tansmit_log...

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(const std::vector<int> & log) {
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(std::vector<int> log) {
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(std::vector<int> log) {
     std::sort( std::begin(log), std::end(log) );
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Ok, ma se fosse un vettore con milioni di elementi?!?!?

- C++11 introduce il concetto di move ownership e rvalue reference

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(std::vector<int> log) {
     std::sort( std::begin(log), std::end(log) );
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(log);
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(std::vector<int> && log) {
     std::sort( std::begin(log), std::end(log) );
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(std::move(log));
}
```

C++ 11 / 14

- `static void transmit_log(`**`std::vector<int> && log`**`)`
  Questo specifica un rvalue reference
- `transmit_log(`**`std::move(log)`**`);`
  Questo, in altre parole, significa: "prendi i dati di questo oggetto, te li regalo, fanne quello che ti pare; prometto che non li userò più, dopo quest istruzione".
- Rvalue reference e la move semantic dovrebbero ridurre il bisogno di creare copie di dati e oggetti nel passaggio di dati alle funzioni permettendo, al contempo, di usare la semantica del call-by-value.
  E' un altro step per nascondere o eliminare del tutto la presenza di puntatori nel codice.

C++ 11 / 14

- Ora, prima di ordinare la lista, vorrei rimuovere tutti gli elementi di log che siano inferiori o uguali a 23.
- Uso la funzione membro erase, applicata a std::remove_if
- ```
  struct filter {
      filter(int limit) : lim(limit) {}
      bool operator()(int i) { return i <= lim; };
      int lim;
  } myfilter(23);
  ```
- ```
  log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
  ```
- Abbiamo creato un oggetto e fatto overload di operatore (), rendendolo un oggetto invocabile (callable). Questi sono detti oggetti funzione o functori.
- Tanto codice inutile! Meglio fare così...

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-
```
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(std::vector<int> && log) {
     std::sort( std::begin(log), std::end(log) );
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(std::move(log));
}
```

C++ 11 / 14

- Per descrivere le novità, partiamo da un esempio...
-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

static void transmit_item(int i) {
     std::cout << i << std::endl;
}

static void transmit_log(std::vector<int> && log) {
     auto myfilter = [limit](int i) {return i <= limit;}
     log.erase(std::remove_if(std::begin(log), std::end(log), myfilter), std::end(log));
     std::sort( std::begin(log), std::end(log) );
     std::for_each(std::begin(log), std::end(log), trasmit_item);
}

int main() {
     std::vector<int> log{20,24,37,42,23,45,37};
     transmit_log(std::move(log));
}
```

C++ 11 / 14

- **Lambda expression**
- Funzioni anonime che mantengono lo stato e che possono avere accesso a tutte le variabili dello scope in cui sono contenute
- Sintassi:
- `[ capture-clause ] ( parameter-list )  -> return-type { lambda-body }`
- `capture-clause` specifica se e come la lambda possa avere accesso alle variabili dello scope che la contiene; può essere lista di variabili già dichiarate.
- Variabili possono essere catturate per valore o per referenza

```
int i = 3;
int j = 5;

// The following lambda expression captures i by value and
// j by reference.
function<int (void)> f = [i, &j] { return i + j; };

// Change the values of i and j.
i = 22;
j = 44;

// Call f and print its result.
cout << f() << endl;
```

- Output: 47

C++ 11 / 14

- **Lambda expression**
- Funzioni anonime che mantengono lo stato e che possono avere accesso a tutte le variabili dello scope in cui sono contenute
- Sintassi:
- `[ capture-clause ] ( parameter-list )  -> return-type { lambda-body }`
- `capture-clause` può anche essere vuota [ ], oppure si possono usare specificatori di default [=] o [&] o un mix di questi:

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

C++ 11 / 14

- **Lambda expression**
- Funzioni anonime che mantengono lo stato e che possono avere accesso a tutte le variabili dello scope in cui sono contenute
- Sintassi:
- `[ capture-clause ] ( parameter-list )  -> return-type { lambda-body }`
- `parameter-list` è opzionale ed ha la stessa struttura e funzione della lista di parametri di una funzione

```
// Assign the lambda expression that adds two numbers to an auto variable.
auto f1 = [](int x, int y) { return x + y; };

cout << f1(2, 3) << endl;
```

C++ 11 / 14

- **Lambda expression**
- Funzioni anonime che mantengono lo stato e che possono avere accesso a tutte le variabili dello scope in cui sono contenute
- Sintassi:
- `[ capture-clause ] ( parameter-list )  -> return-type { lambda-body }`
- `return-type` specifica il return type della funzione lambda. Può essere omesso (di solito è così), a patto che il compilatore possa dedurne il tipo senza ambiguità

C++ 11 / 14

- **Lambda expression**
- Funzioni anonime che mantengono lo stato e che possono avere accesso a tutte le variabili dello scope in cui sono contenute
- Sintassi:
- `[ capture-clause ] ( parameter-list ) -> return-type { lambda-body }`
- `lambda-body` contiene tutto quello che il body di una funzione standard può contenere (incluse dichiarazioni di variabili, chiamate a funzioni, ecc...)
- In più lambda-body ha accesso alle variabili dello scope in cui la lambda è dichiarata, a seconda di come siano state definite dentro la capture list

C++ 11 / 14

- **Lambda expression**
- Esempi:
- Questa lambda cattura esplicitamente la variabile n per valore e implicitamente m per referenza
- Poiché n è catturata per valore, il suo valore non verrà modificato dalla lambda, ma la keyword `mutable` permette di variarne il valore all'interno dello scope
- Output:

```
5
0
```

```cpp
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
   int m = 0;
   int n = 0;
   [&, n] (int a) mutable { m = ++n + a; }(4);
   cout << m << endl << n << endl;
}
```

C++ 11 / 14

- **Lambda expression**
- Esempi:
- E' possibile usare variabili di tipo static all'interno del corpo di una funzione lambda (perchè?)

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

- Questo è anche esempio di come sia possibile definire funzioni lambda all'interno di parameter list di altre funzioni
- Domanda: come scrivere codice equivalente senza usare variabili di tipo static?

C++ 11 / 14

- **Lambda expression**
- Esempi:
- E' possibile usare variabili di tipo static all'interno del corpo di una funzione lambda (perchè?)
- Questo è anche esempio di come sia possibile definire funzioni lambda all'interno di parameter list di altre funzioni
- Domanda: come scrivere codice equivalente senza usare variabili di tipo static?

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

```
int main() {
    std::vector<int> v(10);
    int valore = 0;
    std::generate( std::begin( v ), std::end( v ), [&valore] {return valore++; } );
}
```

C++ 11 / 14

- **Lambda expression**
- Esempi:
- E' possibile usare variabili di tipo static all'interno del corpo di una funzione lambda (perchè?)
- Questo è anche esempio di come sia possibile definire funzioni lambda all'interno di parameter list di altre funzioni
- Domanda: come scrivere codice equivalente senza usare variabili di tipo static?

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this is not thread-safe and is shown for illustration only
}
```

```
int main() {
    std::vector<int> v(10);
    int valore = 0;
    std::generate( std::begin( v ), std::end( v ), [valore] () mutable {return valore++; } );
}
```

C++ 11 / 14

- **Altra bella novità**
- Gestione dei thread ora fa parte del linguaggio! Non è più affidato a librerie che dipendono dal sistema operativo:
  `#include <thread>`
- Suggerimento: date una possibilità alla Microsoft Parallel Programming Library (Windows, Linux, Android)!

```
pplx::task<int> t2( []() {
    std::cout << "IRManager: starting request to Infrastructure Repository" << std::endl;
    return 0;
} );
t2.wait();
auto proc2 = t2
    .then( [=]( int ) { return irm->getPopList();     } )       // Poplist + PoPDetails sequeziali
    .then( [=]( int ) { return irm->getPoPDetails();  } )
    .then( [=]( int ) { return irm->getLinkList();    } )       // linkList e linkDetails sequenziali
    .then( [=]( int ) { return irm->getLinkDetails(); } )
    .then( [=]( int ) { return irm->getHypList();     } )       // hypList e hypDetails sequenziali
    .then( [=]( int ) { return irm->getHypDetails();  } );

//proc1.wait();
proc2.wait();
```

RIPASSO: C

- Bibliografia:
- https://en.wikipedia.org/wiki/Volatile_(computer_programming)
- http://stackoverflow.com/questions/4437527/why-do-we-use-volatile-keyword-in-c
- https://en.wikipedia.org/wiki/Type_qualifier
- http://en.cppreference.com/w/cpp/language/storage_duration
- http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html