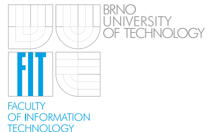
What's New In C++11

Petr Zemek

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 00 Brno, Czech Republic
http://www.fit.vutbr.cz/∼izemek



Decompiler Seminar, 2012-12-04

Contents

- **How Did We Get Here?**
- **What Is New?**
  - Core Language
  - Standard Library
- **What We Have Skipped?**
- **What Has Been Removed?**
- **How Well Is It Supported?**
  - Compilers
  - Standard Libraries
  - Doxygen
- **Discussion**

A Brief History of C and C++

- 1973 – C

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90
- 1998 – ISO C++98

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90
- 1998 – ISO C++98
- 1999 – ISO C99

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90
- 1998 – ISO C++98
- 1999 – ISO C99
- 2003 – ISO C++03

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90
- 1998 – ISO C++98
- 1999 – ISO C99
- 2003 – ISO C++03
- 2007 – ISO C++TR1

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90
- 1998 – ISO C++98
- 1999 – ISO C99
- 2003 – ISO C++03
- 2007 – ISO C++TR1
- 2011 – ISO C++11

A Brief History of C and C++

- 1973 – C
- 1978 – K&R C
- 1981 – C with Classes
- 1985 – C++
- 1990 – ISO C90
- 1998 – ISO C++98
- 1999 – ISO C99
- 2003 – ISO C++03
- 2007 – ISO C++TR1
- 2011 – ISO C++11
- 2011 – ISO C11

C++98

```
1 for (vector<int>::iterator i = v.begin(),
2         e = v.end(); i != e; ++i) {
3     process(*i);
4 }
5
6 std::map<std::string, int>::const_iterator pos =
7     m.find(target);
```

Type Inference (auto)

C++98

```
1 for (vector<int>::iterator i = v.begin(),
2          e = v.end(); i != e; ++i) {
3     process(*i);
4 }
5
6 std::map<std::string, int>::const_iterator pos =
7     m.find(target);
```

C++11

```
8 for (auto i = v.begin(), e = v.end(); i != e; ++i) {
9     process(*i);
10 }
11
12 auto pos = m.find(target);
```

Type Inference (auto)

C++98

```
1 for (vector<int>::iterator i = v.begin(),
2           e = v.end(); i != e; ++i) {
3       process(*i);
4 }
5
6 std::map<std::string, int>::const_iterator pos =
7       m.find(target);
```

C++11

```
 8 for (auto i = v.begin(), e = v.end(); i != e; ++i) {
 9       process(*i);
10 }
11
12 auto pos = m.find(target);
```

Notes: original use of auto

Range-Based For Loop

C++98

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

int numbers[] = {1, 2, 3, 4, 5};
for (size_t i = 0, e = sizeof(numbers)/sizeof(numbers[0]);
         i != e; ++i) {
    numbers[i] *= 2;
}
```

Range-Based For Loop

C++98

```
1  for (vector<int>::iterator i = v.begin(), e = v.end();
2          i != e; ++i) {
3      process(*i);
4  }
5
6  int numbers[] = {1, 2, 3, 4, 5};
7  for (size_t i = 0, e = sizeof(numbers)/sizeof(numbers[0]);
8          i != e; ++i) {
9      numbers[i] *= 2;
10 }
```

C++11

```
11 for (int& x : v) {
12     process(x);
13 }
14
15 for (int& x : numbers) {
16     x *= 2;
17 }
```

C++98

```
1 for (vector<int>::iterator i = v.begin(), e = v.end();
2         i != e; ++i) {
3     process(*i);
4 }
5
6 int numbers[] = {1, 2, 3, 4, 5};
7 for (size_t i = 0, e = sizeof(numbers)/sizeof(numbers[0]);
8         i != e; ++i) {
9     numbers[i] *= 2;
10 }
```

C++11

```
11 for (int& x : v) {
12     process(x);
13 }
14
15 for (int& x : numbers) {
16     x *= 2;
17 }
```

Notes: `auto`, reverse iteration

C++98

```
1 // Create an array of 1, 2, 3, 4, 5.
2 int a[] = {1, 2, 3, 4, 5};
3
4 // Create a vector of 1, 2, 3, 4, 5.
5 std::vector<int> v;
6 v.push_back(1);
7 // ...
8 v.push_back(5);
```

Initializer Lists

C++98

```
1 // Create an array of 1, 2, 3, 4, 5.
2 int a[] = {1, 2, 3, 4, 5};
3
4 // Create a vector of 1, 2, 3, 4, 5.
5 std::vector<int> v;
6 v.push_back(1);
7 // ...
8 v.push_back(5);
```

C++11

```
9 std::vector<int> v = {1, 2, 3, 4, 5};
10
11 void f(std::vector<std::string> v) {
12     // ...
13 }
14 f({"a", "b", "c"});
```

Initializer Lists

C++98

```cpp
1 // Create an array of 1, 2, 3, 4, 5.
2 int a[] = {1, 2, 3, 4, 5};
3
4 // Create a vector of 1, 2, 3, 4, 5.
5 std::vector<int> v;
6 v.push_back(1);
7 // ...
8 v.push_back(5);
```

C++11

```cpp
9 std::vector<int> v = {1, 2, 3, 4, 5};
10
11 void f(std::vector<std::string> v) {
12     // ...
13 }
14 f({"a", "b", "c"});
```

Notes: other STL containers, custom containers

Uniform Initialization

```
1 struct BasicStruct {
2     int x;
3     double y;
4 };
5
6 struct ClassStruct {
7     ClassStruct() {}
8     ClassStruct(int x, double y): x(x), y(y) {}
9     int x;
10    double y;
11 };
```

C++98

```
12 BasicStruct a = {1, 2.5};
13 ClassStruct b(2, 8.9);
14 ClassStruct c((ClassStruct())); // ?
```

```
 1 struct BasicStruct {
 2       int x;
 3       double y;
 4 };
 5
 6 struct ClassStruct {
 7       ClassStruct() {}
 8       ClassStruct(int x, double y): x(x), y(y) {}
 9       int x;
10       double y;
11 };
```

C++98

```
12 BasicStruct a = {1, 2.5};
13 ClassStruct b(2, 8.9);
14 ClassStruct c((ClassStruct())); // ?
```

C++11

```
15 BasicStruct a{1, 2.5};
16 ClassStruct b{2, 8.9};
17 ClassStruct c{ClassStruct()};
```

Alternative Function Syntax + `decltype`

C++11

```
auto f(int x, int y) -> int; // int f(int x, int y);
```

Alternative Function Syntax + decltype

C++11

```
auto f(int x, int y) -> int; // int f(int x, int y);
```

Eh... Why?!

C++11

```
auto f(int x, int y) -> int; // int f(int x, int y);
```

Eh... Why?!

- use in templates

```
1 template<class Lhs, class Rhs>
2 auto add(const Lhs& lhs, const Rhs& rhs)
3     -> decltype(lhs + rhs) {return lhs + rhs;}
```

C++11

```
auto f(int x, int y) -> int; // int f(int x, int y);
```

Eh... Why?!

- use in templates

```
1 template<class Lhs, class Rhs>
2 auto add(const Lhs& lhs, const Rhs& rhs)
3     -> decltype(lhs + rhs) {return lhs + rhs;}
```

- elimination of repetition

```
4 class LongClassName {
5     typedef std::vector<int> IntVec;
6     IntVec f();
7 }
8
9 auto LongClassName::f() -> IntVec { /* ... */ }
10 // vs
11 LongClassName::IntVec LongClassName::f() { /* ... */ }
```

Example

```
[](int x, int y) { return x + y; }
```

Example

```
[](int x, int y) { return x + y; }
```

General Form

```
[capture](parameters) -> return-type { body }
```

Lambda Functions

Example

```
[](int x, int y) { return x + y; }
```

General Form

```
[capture](parameters) -> return-type { body }
```

Use

```
1 std::sort(people.begin(), people.end(),
2           [](const Person& p1, const Person& p2) {
3     return p1.getId() < p2.getId();
4 });
```

Example

```
[](int x, int y) { return x + y; }
```

General Form

```
[capture](parameters) -> return-type { body }
```

Use

```
1 std::sort(people.begin(), people.end(),
2           [](const Person& p1, const Person& p2) {
3     return p1.getId() < p2.getId();
4 });
```

Another Use (Closures)

```
5 std::vector<int> v = // ...
6 int total = 0;
7 std::for_each(v.begin(), v.end(), [&total](int x) {
8     total += x;
9 });
```

Null Pointer Constant

C++98

```
1 void f(int);
2 void f(int *);
3
4 f(0);     // calls void f(int);
5 f(NULL); // also calls void f(int);
```

Null Pointer Constant

C++98

```
1 void f(int);
2 void f(int *);
3
4 f(0);     // calls void f(int);
5 f(NULL); // also calls void f(int);
```

C++11

- a new keyword `nullptr`, type `nullptr_t`
- implicitly convertible only to other pointers and bool

```
6  char *pc = nullptr; // OK
7  int  *pi = nullptr; // OK
8  bool   b = nullptr; // OK (b is false)
9  int    i = nullptr; // !
10
11 f(0);          // calls void f(int);
12 f(nullptr); // calls void f(int *);
```

Strongly Typed Enumerations

C++98

```
1 enum MyEnum { // underlying type is implementation-defined
2     VAL1,
3     VAL2,
4     VAL3
5 };
6
7 int i = VAL2; // implicit conversion, no scoping
```

Strongly Typed Enumerations

C++98

```
1 enum MyEnum { // underlying type is implementation-defined
2       VAL1,
3       VAL2,
4       VAL3
5 };
6
7 int i = VAL2; // implicit conversion, no scoping
```

C++11

```
1 enum class MyEnum: unsigned int { // optional, default int
2       VAL1,
3       VAL2,
4       VAL3, // :)
5 };
6
7 int i = MyEnum::VAL2; // !
```

C++98

```
1 class Base {
2     virtual void f(int);
3 };
4
5 class Derived: public Base {
6     virtual void f(double); // hides Base::f()
7 };
```

C++98

```
1  class Base {
2        virtual void f(int);
3  };
4
5  class Derived: public Base {
6        virtual void f(double); // hides Base::f()
7  };
```

C++11

```
8  class Base {
9        virtual void f(int);
10 };
11
12 class Derived: public Base {
13       virtual void f(double) override; // !
14 };
```

Explicit Overrides

C++98

```
1 class Base {
2     virtual void f(int);
3 };
4
5 class Derived: public Base {
6     virtual void f(double); // hides Base::f()
7 };
```

C++11

```
8 class Base {
9     virtual void f(int);
10 };
11
12 class Derived: public Base {
13     virtual void f(double) override; // !
14 };
```

Notes: `override` is not a reserved word

Final (Classes)

C++98

```
1 class NotBase {}; // Please, do not subclass this class.
2
3 class Derived: public NotBase {};
```

C++98

```
1 class NotBase {}; // Please, do not subclass this class.
2
3 class Derived: public NotBase {};
```

Question: Can this be enforced in C++98?

C++98

```
1 class NotBase {}; // Please, do not subclass this class.
2
3 class Derived: public NotBase {};
```

Question: Can this be enforced in C++98?

C++11

```
4 class NotBase final {};
5
6 class Derived: public NotBase {}; // !
```

Final (Classes)

C++98

```
1 class NotBase {}; // Please, do not subclass this class.
2
3 class Derived: public NotBase {};
```

Question: Can this be enforced in C++98?

C++11

```
4 class NotBase final {};
5
6 class Derived: public NotBase {}; // !
```

Notes: `final` is not a reserved word

Final (Virtual Methods)

C++98

```
1 class Base {
2     virtual void f(); // Please, do not override it.
3 };
4
5 class Derived: public Base {
6     virtual void f();
7 };
```

Final (Virtual Methods)

C++98

```
1 class Base {
2     virtual void f(); // Please, do not override it.
3 };
4
5 class Derived: public Base {
6     virtual void f();
7 };
```

Question: Can this be enforced in C++98?

Final (Virtual Methods)

C++98

```
1 class Base {
2     virtual void f(); // Please, do not override it.
3 };
4
5 class Derived: public Base {
6     virtual void f();
7 };
```

Question: Can this be enforced in C++98?

C++11

```
8 class Base {
9     virtual void f() final;
10 };
11
12 class Derived: public Base {
13     virtual void f(); // !
14 };
```

C++98

```
1 class NonCopyable {
2 private:
3       NonCopyable(const NonCopyable&); // not defined
4       NonCopyable& operator=(const NonCopyable&); // detto
5 };
```

C++98

```
1 class NonCopyable {
2 private:
3     NonCopyable(const NonCopyable&); // not defined
4     NonCopyable& operator=(const NonCopyable&); // detto
5 };
```

C++11

```
6 class NonCopyable {
7 public:
8     NonCopyable(const NonCopyable&) = delete;
9     NonCopyable& operator=(const NonCopyable&) = delete;
10 }
```

Long Long Int

- In C++98 and C90, the largest integer is `long int`.
  - `sizeof(int)` $\leq$ `sizeof(long int)`
  - at least 32b
- In C++11 and C99, the largest integer is `long long int`.
  - `sizeof(long int)` $\leq$ `sizeof(long long int)`
  - at least 64b

Long Long Int

- In C++98 and C90, the largest integer is `long int`.
  - `sizeof(int)` $\leq$ `sizeof(long int)`
  - at least 32b
- In C++11 and C99, the largest integer is `long long int`.
  - `sizeof(long int)` $\leq$ `sizeof(long long int)`
  - at least 64b

```
1 std::numeric_limits<long long int>::min();
2 // At least -9223372036854775808 (2^63).
3 std::numeric_limits<long long int>::max();
4 // At least 9223372036854775807 (2^63 - 1).
5 std::numeric_limits<unsigned long long int>::min();
6 // 0
7 std::numeric_limits<unsigned long long int>::max();
8 // At least 18446744073709551615 (2^64 - 1).
```

C++98

- `assert()` – for run-time assertions
- `#if` + `#error` – for pre-processing assertions
- ? – for compile-time assertions

C++98

- `assert()` – for run-time assertions
- `#if` + `#error` – for pre-processing assertions
- ? – for compile-time assertions

C++11

- `static_assert(const-expr, error-message);`

```
1  template<class Integral>
2  Integral mod(Integral x, Integral y) {
3      static_assert(std::is_integral<Integral>::value,
4          "mod() parameter must be of an integral type");
5      return x % y;
6  }
7
8  mod(1.2, 2.2); // !
9  // In instantiation of ‘Integral mod(Integral, Integral)
10 //     [with Integral = double]’:
11 // error: static assertion failed:
12 //     mod() parameter must be of an integral type
```

- many approaches, only one of them shown

C99/C++98

```
1 int StaticAssertFailed_ExpectedLongToBeAtLeast64b[
2     (sizeof(long) >= 8) ? 1 : -1];
3
4 // error: size of array ‘StaticAssertFailed_
5 //      ExpectedLongToBeAtLeast64b’ is negative
```

Rvalue References

- `&& T`
- bind to temporaries

Rvalue References

- `&& T`
- bind to temporaries

Uses

- move semantics (elimination of spurious copies)

C++98

```
1 template <class T> swap(T& a, T& b) {
2     T tmp(a); // copies a
3     a = b;    // copies b
4     b = tmp;  // copis tmp
5 }
```

- `&& T`
- bind to temporaries

Uses

- move semantics (elimination of spurious copies)

C++98

```
1 template <class T> swap(T& a, T& b) {
2       T tmp(a); // copies a
3       a = b;    // copies b
4       b = tmp;  // copis tmp
5 }
```

C++11

```
6 template <class T> swap(T& a, T& b) {
7       T tmp(std::move(a)); // no copy
8       a = std::move(b);    // no copy
9       b = std::move(tmp);  // no copy
10 }
```

- `&& T`
- bind to temporaries

Uses

- move semantics (elimination of spurious copies)

C++98

```
1 template <class T> swap(T& a, T& b) {
2       T tmp(a); // copies a
3       a = b;    // copies b
4       b = tmp;  // copis tmp
5 }
```

C++11

```
6 template <class T> swap(T& a, T& b) {
7       T tmp(std::move(a)); // no copy
8       a = std::move(b);    // no copy
9       b = std::move(tmp);  // no copy
10 }
```

- perfect forwarding

C++98

```
1 const double d = sqrt(5.6); // run-time evaluation
```

Generalized Constant Expressions

C++98

```
1 const double d = sqrt(5.6); // run-time evaluation
```

C++11

```
2  constexpr double abs(double x) {
3      return (x > 0.0 ? x : -x);
4  }
5
6  constexpr double sqrtImpl(double y, double x1,
7          double x2, double eps) {
8      return (abs(x1 - x2) < eps ? x2 :
9          sqrtImpl(y, x2, (x2 + y / x2) * 0.5, eps));
10 }
11
12 constexpr double sqrt(double y) {
13     return sqrtImpl(y, 0, y * 0.5 + 1.0, 1e-10);
14 }
15
16 const double d = sqrt(5.6); // compile-time evaluation
```

Generalized Constant Expressions

C++98

```
1 const double d = sqrt(5.6); // run-time evaluation
```

C++11

```
2  constexpr double abs(double x) {
3      return (x > 0.0 ? x : -x);
4  }
5
6  constexpr double sqrtImpl(double y, double x1,
7          double x2, double eps) {
8      return (abs(x1 - x2) < eps ? x2 :
9          sqrtImpl(y, x2, (x2 + y / x2) * 0.5, eps));
10 }
11
12 constexpr double sqrt(double y) {
13     return sqrtImpl(y, 0, y * 0.5 + 1.0, 1e-10);
14 }
15
16 const double d = sqrt(5.6); // compile-time evaluation
```

Notes: template meta-programming, turing-completeness

C++98

```
1 std::vector<std::vector<int>> // !
2
3 std::vector<std::vector<int> > // OK
```

Right Angle Brackets

C++98

```
1 std::vector<std::vector<int>> // !
2
3 std::vector<std::vector<int> > // OK
```

C++11

```
4 std::vector<std::vector<int>> // OK
```

Right Angle Brackets

C++98

```
1 std::vector<std::vector<int>> // !
2
3 std::vector<std::vector<int> > // OK
```

C++11

```
4 std::vector<std::vector<int>> // OK
```

Notes: can be overriden by brackets

C++98

```
1 template <typename T>
2 typedef std::map<std::string, T> Dictionary; // !
3
4 // Dictionary<int> d;
```

C++98

```
1 template <typename T>
2 typedef std::map<std::string, T> Dictionary; // !
3
4 // Dictionary<int> d;
```

C++11

```
5 template <typename T>
6 using Dictionary = std::map<std::string, T>; // OK
7
8 Dictionary<int> d;
```

Template Aliases

C++98

```
1 template <typename T>
2 typedef std::map<std::string, T> Dictionary; // !
3
4 // Dictionary<int> d;
```

C++11

```
5 template <typename T>
6 using Dictionary = std::map<std::string, T>; // OK
7
8 Dictionary<int> d;
```

Notes: a new version of `typedef`

C++98

```
 1 template <typename T1>
 2 void print(const T1& val1) {
 3     std::cout << val1 << "\n";
 4 }
 5
 6 template <typename T1, typename T2>
 7 void print(const T1& val1, const T2& val2) {
 8     std::cout << val1;
 9     print(val2);
10 }
11
12 template <typename T1, typename T2, typename T3>
13 void print(const T1& val1, const T2& val2, const T3& val3)
14     std::cout << val1;
15     print(val2, val3);
16 }
17
18 // ...
19
20 print("I am ", 26, " years old."); // I am 26 years old.
```

Variadic Templates (Example)

C++11

```
1  template <typename T>
2  void print(const T& value) {
3      std::cout << value << "\n";
4  }
5
6  template <typename U, typename... T>
7  void print(const U& head, const T&... tail) {
8      std::cout << head;
9      print(tail...);
10 }
11
12 print("I am ", 26, " years old."); // I am 26 years old.
```

- `<tuple>`
- collections composed of heterogeneous objects of pre-arranged dimensions
- access: `get<N>(t)`

- `<tuple>`
- collections composed of heterogeneous objects of pre-arranged dimensions
- access: `get<N>(t)`

Example

```
1 std::tuple<std::string, int> getAddressAndPort(
2         const std::string& url) {
3     // ...
4     return std::make_tuple(address, port);
5 }
6
7 std::string host;
8 int port;
9 std::tie(host, port) = getAddressAndPort("127.0.0.1:631");
```

Tuples

- `<tuple>`
- collections composed of heterogeneous objects of pre-arranged dimensions
- access: `get<N>(t)`

Example

```
1 std::tuple<std::string, int> getAddressAndPort(
2         const std::string& url) {
3     // ...
4     return std::make_tuple(address, port);
5 }
6
7 std::string host;
8 int port;
9 std::tie(host, port) = getAddressAndPort("127.0.0.1:631");
```

Uses

- returning multiple values from a function
- swapping values

```
std::tie(a, b, c) = std::make_tuple(b, c, a);
```

Hash Tables

- `<unordered_{set,multiset,map,multimap}>`
- collisions are managed through linear chaining
- keys have to be hashable and comparable (why?)
- interface is analogous to ordinary `set`s and `map`s

Hash Tables

- `<unordered_{set,multiset,map,multimap}>`
- collisions are managed through linear chaining
- keys have to be hashable and comparable (why?)
- interface is analogous to ordinary `set`s and `map`s

Container	Insert	Erase	Lookup
`set`	$O(log(n))$	$O(log(n))$	$O(log(n))$
`unordered_set`	$O(1)$	$O(1)$	$O(1)$

Hash Tables

- `<unordered_{set,multiset,map,multimap}>`
- collisions are managed through linear chaining
- keys have to be hashable and comparable (why?)
- interface is analogous to ordinary `set`s and `map`s

Container	Insert	Erase	Lookup
`set`	$O(log(n))$	$O(log(n))$	$O(log(n))$
`unordered_set`	$O(1)^*$	$O(1)^*$	$O(1)^*$

* ... average case; $O(n)$ in worst case

Problems with Raw Pointers

- who owns the pointer and should `delete` it?
- when the pointer should be `delete`d?
- they have to be `delete`d manually

Problems with Raw Pointers

- who owns the pointer and should `delete` it?
- when the pointer should be `delete`d?
- they have to be `delete`d manually

Solution: Smart Pointers

```
1 class CarFactory {
2 public:
3     static std::unique_ptr<Car> create(/* parameters */);
4 }
```

Problems with Raw Pointers

- who owns the pointer and should `delete` it?
- when the pointer should be `delete`d?
- they have to be `delete`d manually

Solution: Smart Pointers

```
1 class CarFactory {
2 public:
3     static std::unique_ptr<Car> create(/* parameters */);
4 }
```

C++11 (`<memory>`)

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- ~~`std::auto_ptr`~~ (from C++98, deprecated)

Regular Expressions

- `<regex>`
- default ECMAScript syntax (Javascript style, similar to Perl)
- support for other five syntaxes

Regular Expressions

- `<regex>`
- default ECMAScript syntax (Javascript style, similar to Perl)
- support for other five syntaxes

Algorithms

- `std::regex_match()`
- `std::regex_search()`
- `std::regex_replace()`

- <regex>
- default ECMAScript syntax (Javascript style, similar to Perl)
- support for other five syntaxes

Algorithms

- std::regex_match()
- std::regex_search()
- std::regex_replace()

Example

```
1 void showIPParts(const std::string& ip) {
2     std::regex pattern(
3         R"((\d{1,3}):(\d{1,3}):(\d{1,3}):(\d{1,3}))");
4     std::match_results<std::string::const_iterator> result;
5     bool valid = std::regex_match(ip, result, pattern);
6     if (valid) {
7         std::cout << result[0] << /* ... */ << "\n";
8     }
9 }
```

What Has Been Skipped

- extern templates
- inline namespaces
- modification to the definition of POD
- `sizeof` works on class members without an explicit object
- more keywords: `alignof`, `alignas`, `noexcept`
- allowance of garbage-collected implementations
- support for multi-threaded programming
- unicode string literals (utf-8, utf-16, utf-32)
- user-defined literals
- unrestricted unions
- explicit conversion operators
- extensible random number facility
- wrapper reference
- polymorphic wrappers for function objects
- type traits for meta-programming
- uniform method for computing the return type of functors
- object construction improvement
- ...

What Has Been Removed or Deprecated

Removed:

- `export` (keyword still reserved)

Deprecated:

- dynamic exception specifications
- `std::auto_ptr` (use `std::unique_ptr` instead)
- several other features (function object base classes, adapters, binders)

General Features (1/2)

Language Feature	GCC 4.7	Clang 3.1	VS 2012
Rvalue references			
Initialization of class objects by rvalues			
Non-static data member initializers			
Variadic templates			
Initializer lists			
Static assertions			
auto-typed variables			
Lambda expressions			
Declared type of an expression			
Right angle brackets			
Default template arguments for function templates			
Expression SFINAE			
Alias templates			
Extern templates			
Null pointer constant			
Strongly typed enumerations			
Forward declarations for enums			
Generalized attributes	4.8 yes		
Generalized constant expressions			
Alignment support	4.8 yes		

General Features (2/2)

Language Feature	GCC 4.7	Clang 3.1	VS 2012
Delegating constructors	green	green	red
Inheriting constructors	red: 4.8 yes	red	red
Explicit conversion operators	green	green	red
New character types	green	green	red
Unicode string literals	green	green	red
Raw string literals	green	green	red
Universal character names in literals	green	green	red
User-defined literals	green	green	red
Standard layout types	green	green	green
Defaulted functions	green	green	red
Deleted functions	green	green	red
Extended friend declarations	green	green	green
Extending `sizeof`	green	green	red
Inline namespaces	green	green	red
Unrestricted unions	green	green	red
Local and unnamed types as template arguments	green	green	green
Range-based for	green	green	green
Explicit virtual overrides	green	green	green
Minimal support for garbage collection	red	red	green
Allowing move constructors to throw	green	green	red
Defining move special member functions	green	green	green

Concurrency

Language Feature	GCC 4.7	Clang 3.1	VS 2012
Sequence points			
Atomic operations			
Strong Compare and Exchange			
Bidirectional Fences			
Memory model			
Data-dependency ordering			
Propagating exceptions			
Abandoning a process and `at_quick_exit`			
Allow atomics use in signal handlers			
Thread-local storage			
Dynamic init. and destruction with concurrency			

C99 Features in C++11

Language Feature	GCC 4.7	Clang 3.1	VS 2012
`__func__` predefined identifier			
C99 preprocessor			
long long			
Extended integral types			

libstdc++

- apart from incomplete support of regular expressions and concurrency pretty much OK
- details: `http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html`

libc++

- ? (incomplete support)
- details: `http://libcxx.llvm.org/`

"MS Standard C++ Library"

- ? (incomplete support)
- details: `http://msdn.microsoft.com/en-us/library/vstudio/hh567368.aspx`

What About Doxygen?

Supports C++11 since version 1.8.2 (2012-08).

Supported Features

- strongly typed enums with explicit type
- the `final` keyword on classes and methods
- the `override` keyword for methods
- `nullptr` is new a type keyword in code fragments
- variables with initializer lists
- trailing return types
- template aliases
- variadic templates
- parameters with default lambda functions
- default initializers for non-static data members
- `decltype` as a way selecting a type for a variable
- new string literals
- explictly deleted and defaulted special members

Shameless Advertisement

Petr Zemek: Co je nového v C++11, 4.12.2012

`www.stud.fit.vutbr.cz/~xzemek02/blog/?q=node/116`

Discussion