C++11/14: New opportunities

Marco Foco – C++ User Group Udine 2016/12/03
Optimizations, Improved readability, compile-time correctness

Who am I?

- Marco Foco
  - C++ developer since 1995
  - Working at NVIDIA/Gameworks since October 2015
    - Deep learning engineer, Library designer
  - Past experience
    - Signal & audio/video processing applications
    - Machine learning
    - Automotive
    - Embedded applications
  - Blog: http://marcofoco.com, Twitter: @marcofoco

Why “Opportunities”?

- C++ don’t need new features
  - C++ already works “as is”
  - It’s a Turing-complete language
  - It’s sufficiently low-level to allow an efficient implementation of most common problems
- Some features could be unavailable
  - Lack of compiler support
  - Working in low resource environments
- Specific coding rules for a project/team

Which Opportunities?

- Enforcing compile-time correctness
  - Testing is good, catching errors before tests is even better!
- Improving readability
  - “Always code as if the person who end up maintaining your code is a violent psychopath”
- Enabling new optimizations
  - Why not?
- Increasing productivity
  - Be faster in prototyping new ideas

Language Improvements

- Type-safety improvements
- Compile-time evaluation and error generation
- Type deduction
- r-value references
- Move semantics
- Inline function object declaration (lambda)

C++11/14: New opportunities

Final & Override

Override & Final specifiers

- New keywords! Can be used to control overriding and inheritance.

```
namespace override {
    class override {};
    class override_final {
        ::override::override override;
    public:
        virtual ::override::override & final() { return override; }
    };
}
namespace final {
    class final final : public override::override {};
    class final_override final : public override::override_final {
        ::final::final override;
    public:
        ::final::final & final() override { return override; }
    };
}
```


Don’t worry, they’re context sensitive:

override (1)

- Explicitly declares that a method is overriding a virtual method from one of its superclasses

```
class A {
public:
    virtual void a();
};

class B : public A {
public:
    void a() override;
};
```

- It's convenient to apply this change as first operation before a code refactoring/ update

override – a real case (1)

- Library code (version 1)

```
namespace lib {
    class A {
    public:
        virtual void recv();
...
    };
}
```

override – a real case (2)

- Application code (C++03) overriding recv method

```
namespace app {
    class B : public lib::A {
    public:
        void recv() {
            cout << “data received!" << endl;
        }
    };
}
```

override – a real case (3)

- Library code (version 2)

```
namespace lib {
    class A {
    public:
        virtual void recv(int count);
...
    };
}
```

override – a real case (4)

- What happened?
- `recv()` method is no longer called
- You can still blame the library maintainer, but that won't help solving the bug.

override – a real case (5)

- Application code (C++11)
- Marking recv method as override

```
namespace app {
    class B : public lib::A {
    public:
        void recv() override {
            cout << "data received!" << endl;
        }
    };
}
```

override – a real case (6)

- Upgrading the library to version 2 after upgrading app's codebase to use C++11's override:

```
main.cpp:15:30: error: non-virtual member function
marked 'override' hides virtual member function
        void recv() override {
                    ^
library.h:5:22: note: hidden overloaded virtual
function 'lib::A::data_received' declared here:
different number of parameters (1 vs 0)
        virtual void recv(int amount)
                     ^
```

final

- Disallow inheritance from this class

```
class A final {…}
```

- Disallow overriding of a virtual method

```
class A {
    public:
    virtual void a() final;
};
```

final – sealing classes (1)

- Example:

```
class A final {};
class B : public A {};
```

- Compiler result:

```
main.cpp:2:18: error: base 'A' is marked 'final'
class B : public A {};
                 ^
main.cpp:1:7: note: 'A' declared here
class A final {};
      ^ ~~~~~
```

final – sealing classes (2)

- C++03 equivalent code is not as readable

```
class final_helper_for_A {
    friend class A;
private:
    ~final_helper_for_A() {}
};

class A : virtual public final_helper_for_A {};
class B : public A {};
```

- CAVEAT: a virtual base class (public final_helper_for_A) is always treated as a direct base of its most derived class (B)

final – sealing classes (3)

- Error messages in C++03 were even worse
- Also, they're produced on the class usage, not at declaration

```
main.cpp:11:3: error: call to implicitly-deleted
default constructor of 'B'
B b;
  ^
main.cpp:7:11: note: default constructor of 'B' is
implicitly deleted because base class
'final_helper_for_A' has an inaccessible destructor
class A : virtual public final_helper_for_A {};
          ^
```

final – sealing methods (1)

```
class A {
public:
    virtual void a();
};

class B : public A {
public:
    void a() final;
};

class C : public B {
    void a();
};
```


Note: final implies override (but it's not an error to add both)

final – sealing methods (2)

- Compiler result:

```
main.cpp:12:7: error: declaration of 'a'
overrides a 'final' function
        void a();
             ^
main.cpp:8:7: note: overridden virtual
function is here
        void a() final;
             ^
```


There's no way to achieve the same behavior in C++03

override & final support

- C++11 (N2928, N3272, N3272)
  - Clang 3.0 (Dec 2011)
  - GCC 4.7 (Mar 2012)
  - Visual Studio 2012 (Sep 2012)

C++11/14: New opportunities

nullptr

nullptr (1)

- Improved, type safe version of `0` or `NULL` when referring to pointers
- `nullptr`'s type is a new type, `nullptr_t`
- Implicitly converts only to pointer types (no risk of taking into account int overloads during overload resolution)
- Standard (`NULL` is standard in C99, but expands to a null pointer, so it might expand to `0`)

nullptr (2)

- Example

```
void a(int x) { cout << "int" << endl; }
void a(void *y) { cout << "pointer" << endl; }

int main() {
    a(0);
    a(NULL); // platform-dependent!
    a(nullptr);
}
```

- Output (VS2015/x86 & x64)

```
int
int
pointer
```

nullptr (3)

- The definition of `NULL` is platform-specific
  - The value is 0, but the type may vary
    - Visual Studio (2015): it's an int, so the code on the previous slide compiles and invokes the int overload
    - Clang (3.7 x64): it's a long, so the code above gives an error:

```
error: call to 'a' is ambiguous
```


Clang would have matched a long overload, if present, but any other overload requires an implicit conversion

nullptr support

- C++11 (N2431)
  - Visual Studio 2010 (Apr 2010)
  - GCC 4.6 (Mar 2011)
  - Clang 3.0 (Dec 2011)

C++11/14: New opportunities

constexpr

constexpr (C++11)

- New keyword (NOT context-sensitive)
  - old code using `constexpr` as a name is going to fail when using new C++ standard(s)
- Used in
  - Variable declaration
  - Constructor, method or function definition

constexpr variables (1)

- Prefixing a variable definition with `constexpr` declares the variable as being a compile time constant
  - A compile time constant can be used everywhere a literal value (e.g. `5`) can be used
- Its type must be a literal type
  - A literal type is, a type that can be initialized and manipulated via `constexpr` constructor and functions
- `constexpr` variables might be allocated in a constant segment

constexpr variables (2)

- Define a compile-time constant

```
constexpr int C = 5;
```

- C can be used in all places where the literal 5 was used
  - As size for a statically-sized array

```
int arr[C];
```

  - As template parameter

```
std::array<int, C> arr;
```

constexpr functions (1)

- Define a custom function (or constructor or method) capable of being used in `constexpr` evaluations
- If used in a constructor, allow the type to be a literal type

constexpr functions (2)

- The function might be evaluated at compile time in order to generate a compile time constant
- It will be evaluated at compile time when a literal value is required, or when defining a constant expression variable
- It will execute at run-time when parameters are non-literal and/or non constant expression values

constexpr functions (3)

- When a `constexpr` function is evaluated at compile-time
  - all code that is involved in the computation must follow the rules for `constexpr` functions.
  - Code that is not involved in the computation may contain other statements
- Different rules for C++11 and C++14
  - C++11 `constexpr` functions are still valid in C++14

constexpr functions (4)

- The C++11 standard requesting the function to only contain
  - Null statements (`;`)
  - `static_assert`s
  - `using`s
  - `typedef`s
- And to consist of a single return statement

constexpr functions (5)

- Example (C++11):

```
constexpr int factorial(int n) {
    return n <= 1? 1 : (n * factorial(n - 1));
}

template <int N> struct X {
    void printN() { cout << N << endl; }
};

int main() {
    constexpr int a = factorial(5); // 120
    int q[a];
    cout << sizeof(q) << endl;
    X<a> x;
    x.printN();
}
```

constexpr functions (6)

- The new C++14 standard lists the forbidden statements/constructs in a `constexpr` function
  - No `asm` declarations
  - No `goto` statements
  - No `try` blocks
  - No definition of non-literal typed variables
  - No `static` or `thread_local` variables
  - All variables must be initialized

constexpr functions (7)

- Example (C++14):

```
constexpr int factorial(int n) {
    int result = 1;
    for(int i = 2; i <= n; ++i)
        result *= i;
    return result;
}
```

constexpr support

- C++11 (N2235)
  - Visual Studio 2015 (Jul 2015)
  - GCC 4.6 (Mar 2011)
  - Clang 3.1 (May 2012)
- C++14 (N3652)
  - Clang 3.4 (Jan 2014)
  - GCC 5.1 (Apr 2015)
  - Visual Studio: Not supported (2017?)

C++11/14: New opportunities

static_assert

static_assert (1)

- Compile-time checks, with error message

```
static_assert(sizeof(int) >= 4, ”small");
```

- Can be used to run unit test on `constexprs` at compile time

```
static_assert(factorial(5) == 120, ”bug");
```


Can be used in templates, and to do compile-time checks
The condition must always be a constant expression

static_assert (2)

- Bad interaction within **constexpr** functions

```
constexpr int double_if_small(int n) {
  static_assert(n < 10, "too big");   // fail
  return n * 2;
}
```

- This can be solved with a (quite bad) hack

```
constexpr int double_if_small(int n) {
  return (n < 10)?n * 2:throw "too big";
}
```


This code fails to compile because in non-constexpr context, n is not a constant expression.

static_assert support

- C++11 (N1720)
  - Visual Studio 2010 (Apr 2010)
  - GCC 4.3 (Mar 2008)
  - Clang 2.9 (Apr 2011)

C++11/14: New opportunities

using

using

- New use to the keyword `using`, similar to `typedef`, but with a cleaner syntax

`using myint = int;` Is equivalent to `typedef int myint;`

- It also allows template type definitions

```
template<typename T> using stringmap = map<string, T>;
```

- Also allow to create definitions for function types

```
using myfun = int(int);
```

using – support

- C++11 (N2258)
  - Clang 3.0 (Dec 2011)
  - GCC 4.7 (Mar 2012)
  - Visual Studio 2013 (Oct 2013)

C++11/14: New opportunities

Type deduction

Type deduction

- Letting the compiler figure out the type for a variable, as opposed to (explicit) types declaration.
  - Types are deduced at compile time
  - It's also diffuse in other languages, such as C#, ML, Haskell...
  - It's useful both for generic programming and for lazy programmers
  - Improves readability by removing complex types

auto

- Follows the same type-deduction rules used for templates
  - Strips references and cv-qualifiers
- Variables (since C++11)

```
auto x = 100;  // x is int
auto y = 10.0; // x is double
vector<map<string, int>> arr;
auto z = arr.begin();
  // z is vector<map<string, int>>::iterator_type
```

auto as return type

- Trailing return types for function (C++11)

```
auto f(int x) -> int { return x*2; }
```

- Return type deduction for functions (C++14)

```
auto f(int x) { return x*2; }
  // return type is int
```

decltype

- In C++11 another kind of deduction is obtaining the type from an expression

```
decltype(x) y = x; // y has the same type as x
```

- For function it's possible to use **decltype**, too:

```
auto f(int x) -> decltype(x*2.0) { return
x*2.0; }
```

- In this case, the return type is deduced from the expression inside the decltype

WARNING: decltype behavior is different from auto with respect to references and cv-qualifiers
decltype also deduces references and cv-qualifiers

decltype(auto)

- Introduced in C++14
  - Can be used in function and variable definition
  - Deduces the types (as `auto`)
  - Uses type deduction rules from `decltype`
    - retain references and cv-qualifiers

Type deduction support

- C++11 (N1984)
  - Visual Studio 2010 (Apr 2010)
  - GCC 4.4 (Mar 2009)
  - Clang 2.9 (Apr 2011)
- C++14 (N3638)
  - Visual Studio 2015 (Jul 2015)
  - GCC 4.9 (Apr 2014)
  - Clang
    - 3.3 (Jun 2013) – `decltype(auto)`
    - 3.4 (Jan 2014) – return type deduction

C++11/14: New opportunities

rvalues and move semantics

rvalue references

- New type of references, representing a reference that cannot be used on the left side of an assignment.
- Used to represent reference to temporary objects (or unnamed variables)
- Qualified using `&&`

```
int &&x = …;
```

rvalue conversions

- `&&` is not the same as `const &`
  - `&&` can be implicitly converted into `const &`
  - `const &` can't be implicitly converted to `&&`
  - `const &&` and is equivalent to `const &`, but it's a distinct type
  - `const &&` can't be used in move semantics

Const && anc const & are still dinstinct types

overload with ref qualifiers

```
class C {
    string content;
    string &get() & { return content; } // 1
    string get() && { return content; } // 2
};

C f() { return C(); }

int main() {
    C c;
    auto x = c.get();   // 3: will invoke 1
    auto y = f().get(); // 4: will invoke 2
}
```


Gives the possibility to overload a method with reference qualifier, to differentiate between the contexts where the method is used
If we remove 2, we explicitly forbid calling get() on a temporary object, and line at 4 will generate an error.

Move semantics (1)

- The move constructor

```
class X {
    X(X&& x) {…}
};
```

- "Steals" resources from the original object, which is left in an unspecified state
- `const &&` are pointless

In C++11, variables going out of scope (in the return line) and temporaries returned by value are treated as rvalue references
This enable a new form of constructor to be activated when returning (complex) objects:

The move constructor usually "steals" the resources from the dying object and "move" them to the destination, rather than copying them

Move semantics (2)

- Move assignment

```
class X {
    X& operator = (X&& x) {…}
};
```

- Move constructors can be elided too

As in the copy-constructor <-> copy-assignment duality, move-constructor have its dual, the move-assignment

Move semantics (3)

```
struct Tr {
    Tr() { cout << "cons" << endl; }
    Tr(int n) { cout << "cons-" << n << endl; }
    Tr(const Tr&) { cout << "copycons" << endl; }
    Tr & operator = (const Tr&) {
        cout << "copyassign" << endl;
        return *this;
    }
    ~Tr() { cout << "dest" << endl; }
};
```


Constructor tracking class in C++03

Move semantics (4)

```
int main() {
    Tr t(1);
    t = Tr();
    cout << "end" << endl;
}
```

- Output (C++03)

```
cons-1
cons
copyassign
dest
end
dest
```

Move semantics (5)

- Constructor tracking class in C++11

```
struct Tr {
    // as before
    Tr(Tr&&) { cout << ”movecons" << endl; }
    Tr & operator = (Tr&&) {
        cout << ”moveassign" << endl;
        return *this;
    }
};
```

Move semantics (6)

- Output (C++11)

```
cons-1
cons
moveassign
dest
end
dest
```


We didn't have to change the test code in order to take advantage of the move semantics!

Move semantics (7)

- Allows to write efficient functions that return complex objects

```
Tr f() {
    Tr v(100);
    // Some complexity and other return
    // statements here, to avoid NRVO
    return v;
}
```


In the return line v is automatically converted into a rvalue reference, thus allowing move constructor and assignment operators to be used in this context

Creating a good example for this is complex, because in simple cases the return value optimization kicks in, and the copy/move constructor is optimized away.

RVO/NRVO will be mandatory in C++17

Move semantics (8)

- Explicit moves (using **`std::move`**)

```
int main() {
    Tr t(1);
    t = Tr();
    Tr t2 = move(t);
    cout << "end" << endl;
}
```


T is left in an unspecified state, but it will correctly destroy when the destructor is called at the end of scope

Move semantics (9)

- Output (C++11)

```
cons with 1
cons
moveassign
dest
movecons
end
dest
dest
```

Move semantics (10)

- What if I use move on the old C++03 version?

```
cons with 1
cons
copyassign
dest
copycons
end
dest
dest
```

Move semantics (11)

- What really is std::move?
  - It's a cast operator, from `&` to `&&`
- Why `std::move` performed a copy in the previous example?
  - Since no constructor matched `Tr&&`, the rvalue reference got implicitly converted back into a `const Tr&`, and the copy constructor got invoked
- That's another reason why you can't make any assumption on the state of the variable
  - moved objects might not move at all!

Extended references

- xref a.k.a. Universal references (Meyers)
- Syntactically they're rvalue references of a generic type

```
auto&& something = …;
template<typename T> void f(T&& t) {…}
```

- Can either be an lvalue or rvalue reference
- Useful to forward the exact type to a function or constructor (exact references included)

rvalue and move support

- C++11 (N1610, N2118, N2439)
  - Clang 2.9 (Apr 2011)
  - Visual Studio
    - Partial support since Visual Studio 2010 (Apr 2010)
    - Full support since Visual Studio 2015 (Jul 2015)
  - GCC
    - N1610: All versions (as custom extension)
    - N2118: 4.3 (Mar 2008)
    - N2439: 4.8.1 (May 2013)

C++11/14: New opportunities

noexcept

noexcept (1)

- Used as a specifier, (conditionally) disables exceptions

```
void f() noexcept {…}
void f() noexcept(condition) {…}
```

- Used as an operator

```
constexpr bool b = noexcept(expression);
```


Specifier
Replaces throw(), with a more efficient semantics
Reduces the EH overhead to zero
If an exception is raised, the execution halts
Can be conditionally enabled

Used as an operator, Returns true if the expression is noexcept

noexcept (2)

- Example

```
int mult(int m1, int m2) noexcept {
    return m1*m2;
}
```

- Conditional noexcept is useful when dealing with generic code

```
template<typename T>
T mult(T m1, T m2) noexcept(noexcept(m1*m2)) {
    return m1*m2;
}
```


If the function throws, unexpected() is called.
The above code is noexcept for ints, but could be throwing when other types are used (e.g a Matrix class, where multiplication throws on matrix dimension mismatch)

noexcept support

- C++11 (N3050)
  - GCC 4.6 (Mar 2011)
  - Clang 3.0 (Dec 2011)
  - Visual Studio 2015 (Jul 2015)

C++11/14: New opportunities

Lambdas

Stateless lambda

- Inline definition of functions

```
auto f = [](int x) { return x*2; }
```

- Good for small computation (e.g. for <algorithm>s)

```
std::sort(x.begin(), x.end(), [](T x, T y) {
    // perform custom comparison with x and y
});
```

- Can be assigned to c-style function pointers

Are preferable for small computations to be used as customization points for function
e.g. functions from the <algorithm> header

Stateful lambda (1)

- Inline definition of function objects

```
int y = 5;
auto f = [y](int x) { return x*y; }
```

- An object is created behind the scene

```
class <nameless> {
    int _y;
    <nameless>(int y) : _y(y) {}
    auto operator()(int x) {
        return x*_y;
    }
};
```


Can capture variables from the current scope, y is the captured variable (by value, in this case)

Stateful lambda (2)

- Capturing all by value

```
int y = 5;
auto f = [=](int x) { return x*y; }
```

- Capturing all by reference

```
int y = 5;
auto f = [&](int x) { return x*y++; }
```

- Capturing specific variables by reference or by value

```
int y = 5, z = 7;
auto f = [&y, z](int x) { return z*x*y++; }
```

Stateful lambda (3)

- Capturing by value and modifying the copy (C++14)

```
int y = 5;
auto f = [y](int x) mutable { return x*y++; }
```

- Moving objects into lambdas (C++14)

```
complex_type t;
auto f = [q = move(t)](int x) { …use of q… }
```


Improvements in C++14
Generic computations in the capture list are supported, too, but strongly discouraged.

Stateful lambda (4)

- How to pass stateful lambdas to a function?

```
void function g(<type?> f) {
    return f(1);
}
g([y](int x) { return x*y; });
```

- Make it a generic function and use extended references

```
template <typename T> void function g(T &&f) {
    return f(1);
}
```

- Use std::function

```
void function g(const function<int(int)> &f) {
    return f(1);
}
```

Generic Lambda (C++14)

- Lambdas using auto as parameter

```
auto f = [](auto x) { return x*2; }
```

- Are equivalent to template functions (stateless) or or templated function object (stateful)

```
template<typename T> auto f(T x) {
    return x*2;
}
```

Generic Lambda (C++14)

- Lambdas using auto as parameter

```
auto f = [](auto x) { return x*2; }
```

- Are equivalent to template functions (stateless) or or function object with a templated `operator()` (stateful)*

```
template<typename T> auto f(T x) {
    return x*2;
}
```

* Special thanks to Nicola Gigante for pointing this out

Lambda support

- C++11 (N2927)
  - GCC 4.5 (Apr 2010)
  - Visual Studio 2012 (Aug 2012)
    - N2658 (pre-standard) in Visual Studio 2010 (Mar 2010)
  - Clang 3.1 (Dec 2012)
- C++14 (N3648, N3649)
  - Clang 3.4 (Jan 2014)
  - GCC 4.9 (Apr 2014)
  - Visual Studio 2015 (Jul 2015)

C++11/14: New opportunities

Conclusions

Conclusions (1)

- `override` & `final`
  - Improve readability by documenting the class
  - Enforce compile-time correctness of overridden and non-overridable methods
- `nullptr`
  - Improve readability by making programmer's intention clear
  - Improve compile-time correctness by removing spurious overloads from overload resolution

Conclusions (2)

- `constexpr`
  - Improve readability by removing magic constants, macros and/or template meta-programming black magic
  - It's a possible source of optimization by allowing pre-computation of complex structures
- `static_assert`
  - Enforce compile-time correctness
  - Improves readability of error messages

Conclusions (3)

- `using`
  - Improve readability reducing long type names
  - Increase productivity
- Type deduction
  - Improve readability removing long type names
  - Remove sources of errors, such as mistyping type names, and performance bugs, such as unwanted implicit castings silently generated by mismatching types
  - Increase productivity

Conclusions (4)

- rvalue references & move semantics
  - Enable new optimisations for moving objects around
  - Improve readability by allowing the use of function return rather than reference-passing output variables
- `noexcept`
  - Enable new optimizations
  - Improve productivity by allowing exception mechanism in the non performance-critical code

Conclusions (5)

- lambda
  - Improve readability by improving locality of source code (you no longer need to declare elsewhere small functions)
  - Improve productivity by simplifying the introduction of customization points on algorithms

Questions