

Support for Embedded Programming in C++11 and C++14



Photo: Craig Wyzlk @ flickr

Scott Meyers, Ph.D.

Copyrighted material, all rights reserved.
Last Revised: 10/24/14

Features New in C++11 (Incomplete List)

auto	explicit conversion functions	alias templates
range-based for	override, final	move semantics
scoped enums	underlying type for enums	static_assert
smart pointers	make_shared, allocate_shared	variadic templates
lambda expressions	enhanced const_iterator support	std::bind
type traits	numeric to string conversions	tuples
alignment control	>> as nested template closer	regular expressions
user-defined literals	default member initializers	singly-linked lists
hash tables	support for emplacement	new algorithms
perfect forwarding	support for concurrency	enable_if
constexpr	support for Unicode	rvalue references
raw string literals	uniform (braced) initialization	universal references
std::array	inherited & delegated constructors	nullptr
initializer_list	std::function	defaulted functions
decltype	trailing return types	deleted functions

Features New in C++14 (Incomplete List)

Generalized function return type deduction	<code>make_unique</code>
Generic lambdas	Non-member <code>cbegin</code> , <code>cend</code> , <code>rbegin</code> , <code>rend</code> , <code>crbegin</code> , <code>crend</code>
Lambda init capture	Literal suffixes for <code>string</code> , <code><chrono></code> units, complex numbers
Relaxed rules for <code>constexpr</code> functions	Readers-Writer locks
Binary literals	IO support for quoted strings
Single quote as digit separator	Type-based <code>tuple</code> access
Variable templates	Aliases for transformation traits
Global operator delete with size parameter	Simplified operator functor templates
<code>[[deprecated]]</code> attribute	

C++11 and C++14 Features

Most as useful for embedded as for non-embedded development.

Covering—even *mentioning*—everything not possible.

Features New in C++11 (Incomplete List)

<code>auto</code>	explicit conversion functions	alias templates
range-based for	<code>override final</code>	move semantics
scoped enums	underlying type for enums	<code>static_assert</code>
smart pointers	<code>make_shared</code> , <code>allocate_shared</code>	variadic templates
lambda expressions	enhanced <code>const_iterator</code> support	<code>std::bind</code>
type traits	<code>numeric</code> ⇌ <code>string</code> conversions	tuples
alignment control	<code>>></code> as nested template closer	regular expressions
user-defined literals	default member initializers	singly-linked lists
hash tables	support for emplacement	new algorithms
perfect forwarding	support for concurrency	<code>enable_if</code>
<code>constexpr</code>	support for Unicode	rvalue references
raw string literals	uniform (braced) initialization	universal references
<code>std::array</code>	inherited & delegated constructors	<code>nullptr</code>
<code>initializer_list</code>	<code>std::function</code>	defaulted functions
<code>decltype</code>	trailing return types	deleted functions

Features New in C++14 (Incomplete List)

Generalized function return type deduction	<code>make_unique</code>
Generic lambdas	Non-member <code>cbegin</code> , <code>cend</code> , <code>rbegin</code> , <code>rend</code> , <code>crbegin</code> , <code>crend</code>
Lambda init capture	Literal suffixes for <code>string</code> , <code><chrono></code> units, complex numbers
Relaxed rules for <code>constexpr</code> functions	Readers-Writer locks
Binary literals	IO support for quoted strings
Single quote as digit separator	Type-based <code>tuple</code> access
Variable templates	Aliases for transformation traits
Global operator delete with size parameter	Simplified operator functor templates
<code>[[deprecated]]</code> attribute	

Agenda

auto

Enhanced enums
(if time allows)

Alignment control
(in Bartek Szurgot's talk)

constexpr

Features New in C++11 (Incomplete List)

auto	explicit conversion functions	alias templates
range-based for	override final	move semantics
scoped enums	underlying type for enums	static_assert
smart pointers	make_shared allocate_shared	variadic templates
lambda expressions	enhanced const_iterator support	std::bind
type traits	numeric::string conversions	tuples
alignment control	>> as nested template closer	regular expressions
user-defined literals	default member initializers	singly-linked lists
hash tables	support for emplacement	new algorithms
perfect forwarding	support for concurrency	enable_if
constexpr	support for Unicode	rvalue references
raw string literals	uniform (braced) initialization	universal references
std::array	inherited & delegated constructors	nullptr
initializer_list	std::function	defaulted functions
decltype	trailing return types	deleted functions


Features New in C++14 (Incomplete List)

Generalized function return type deduction	make_unique
Generic lambdas	Non-member cbegin, cend, rbegin, rend, crbegin, crend
Lambda init capture	Literal suffixes for string, <chrono> units, complex numbers
Relaxed rules for constexpr functions	Readers-Writer locks
Binary literals	IO support for quoted strings
Single quote as digit separator	Type-based tuple access
Variable templates	Aliases for transformation traits
Global operator delete with size parameter	Simplified operator functor templates
[[deprecated]] attribute	

Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.
Slide 5

auto



Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.
Slide 6

Prefer auto to Explicit Type Declarations

Given

```
Widget w = expression of type Widget;
```

```
auto w = expression of type Widget;
```

why prefer auto?

Prefer auto to Explicit Type Declarations

Mandatory initialization:

```
int sum; // possibly uninitialized
```

```
auto sum; // error! no initializer from which  
// to deduce type
```

```
auto sum = 0; // fine, sum is int initialized to 0
```

Prefer auto to Explicit Type Declarations

Typically less verbose; avoids syntactic noise:

```
std::vector<std::string::iterator> vsi;
...
for (std::string::iterator si : vsi) ...      // explicit type
for (auto si : vsi) ...                      // auto

template<typename It>
void someAlgorithm(It b, It e)
{
    typename std::iterator_traits<It>::value_type // explicit
        firstElem = *b;                          // type
    auto firstElem = *b;                          // auto
    ...
}
```

Prefer auto to Explicit Type Declarations

Avoids “type shortcut”-related problems.

```
std::vector<int> v;
...
unsigned sz = v.size();    // type “shortcut”!
                        // (should be
                        // std::vector<int>::size_type)

▪ Fine on 32-bit Windows.
  ▶ unsigned and std::vector<int>::size_type are 32 bits.

▪ Unreliable on 64-bit Windows.
  ▶ unsigned is 32 bits, std::vector<int>::size_type is 64 bits.

auto sz = v.size();      // sz is std::vector<int>::size_type
```

Example due to Andrey Karpov.

Prefer auto to Explicit Type Declarations

Avoids accidental temporary creation:

```
std::map<std::string, int> m; // holds objects of type
                             // std::pair<const std::string,
                             // int>
...
for (const std::pair<std::string, int>& p : m) ...
    // creates temp on
    // each iteration =>
    // std::string is copied
for (const auto& p : m) ... // no temps created
```

Example due to Stephan T. Lavavej.

Prefer auto to Explicit Type Declarations

Most efficient way to store function objects:

```
std::function<int(int)> f1a = // closure could
[a,b](int x) { return x*x - a/10 + b; }; // be on heap

auto f1b = // closure not on
[a,b](int x) { return x*x - a/10 + b; }; // heap
```

Prefer auto to Explicit Type Declarations

Efficiency difference not limited to closures:

```
inline void f(int x, int y, int z) { ... }
std::function<void()> f2a =      // func. obj. from std::bind
    std::bind(f, a, b, c);      // could be on heap

auto f2b = std::bind(f, a, b, c); // func. obj. from std::bind
// not on heap
```

auto and Code Clarity

auto a tool, not an obligation.

- If explicit type clearer, use it.

But:

- IDEs may show types of auto-declared variables.
 - ➔ E.g. Visual Studio 2010+.
- Much success in other languages with similar features.

auto



constexpr



Use constexpr Whenever Possible

Two meanings:

- **Objects:** const + value known during compilation.
- **Functions:** \approx return constexpr result if called with constexpr args.
 - ➔ Truth a bit more subtle.

constexpr Objects

Must be initialized with value known during compilation:

```
constexpr std::size_t arraySize;    // error! no initializer
int sz;                             // non-constexpr variable
...
constexpr auto arraySize1 = sz;     // error! sz's value not
                                   // known at compilation
constexpr auto arraySize2 = 10;     // okay
```

Usable in contexts requiring compile-time constants:

```
std::array<int, sz> data1;           // error! sz's value still
                                   // unknown at compilation
std::array<int, arraySize2> data2;  // fine, arraySize2
                                   // is constexpr
```

const (hence unmodifiable):

```
arraySize2 = 100;                   // error!
```

constexpr Objects vs. const Objects

const objects may be initialized at runtime:

```
int sz; // as before
...
const auto arraySize3 = sz; // fine, arraySize3 is
                           // const copy of sz
```

Such objects invalid in contexts requiring compile-time values:

```
std::array<int, arraySize3> data3; // error! arraySize3
                                   // isn't constexpr
```

All constexpr objects are const, but not all const objects are constexpr.

constexpr Objects vs. const Objects

Bottom line:

- Need compile-time value ⇒ constexpr is proper tool.
- When constexpr viable, use it.
 - ➔ Resulting object usable in more contexts.

```
const int maxVal = 100; // typically inferior
constexpr int maxVal = 100; // typically superior
```

constexpr Functions

Quite different from `const` functions.

- Not limited to member functions.
- “Execute” during compilation and return `constexpr` results if :
 - ➔ Arguments are `constexpr`.
 - ➔ Result used in `constexpr` context.
- *May* (but *may not*) “execute” during compilation if:
 - ➔ Arguments are `constexpr`.
 - ➔ Result *not* used in `constexpr` context.
- Execute at runtime for ≥ 1 non-`constexpr` argument.
 - ➔ No need to “overload” on `constexpr`.
 - ◆ `constexpr` functions take `constexpr` *and* non-`constexpr` args.

Note:

- `constexpr` functions may not return `const` results.
- `constexpr` functions may not “execute” during compilation.

constexpr Functions

Consider an experiment where:

- 3 possible lighting levels: high, low, off.
- 3 possible fan speeds: high, low, off.
- 3 possible temperature values: high, medium, low.
- Etc.

n independent 3-valued conditions $\Rightarrow 3^n$ combinations.

Holding all results in a `std::array` \Rightarrow compute 3^n during compilation.

- Goal: calculate with `constexpr` function.

constexpr Functions

Client view:

```
constexpr int pow(int base, int exp) noexcept // constexpr
{                                             // function
    ...                                       // coming soon
}
constexpr auto numConds = 5;
std::array<int, pow(3, numConds)> results; // fine, results
                                           // has 3^numConds
                                           // elements
```

Note:

- pow called with constexpr values in a constexpr context.
 - ➔ Guarantees compile-time evaluation.

constexpr Functions

pow also callable at runtime:

```
auto base = readFromDB("base"); // runtime call
auto exp = readFromDB("exponent"); // runtime call
auto baseToExp = pow(base, exp); // runtime call
```

constexpr Functions

constexpr functions are constrained.

- May take and return only *literal types*.
 - ➔ Essentially types where values compile-time-computable.
- Implementations restricted.
 - ➔ Restrictions differ for C++11 and C++14.

In C++11, exactly one executable statement: `return`.

- But `?:` emulates `if/else` and recursion emulates iteration, so:

```
constexpr int pow(int base, int exp) noexcept
{
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

constexpr Functions

C++14 restrictions much looser.

- Multiple statements okay
- Local variables of literal type okay.
- Iteration statements okay.
- Etc.

Hence:

```
constexpr int pow(int base, int exp) noexcept           // C++14
{                                                       // only
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

Literal UDTs

Literal types can be user-defined!

- constexpr constructors and other member functions.

```
class Point {
public:
    constexpr Point(double xVal = 0, double yVal = 0) noexcept
        : x(xVal), y(yVal)
    {}

    constexpr double xValue() const noexcept { return x; }
    constexpr double yValue() const noexcept { return y; }

    void setX(double newX) noexcept { x = newX; }
    void setY(double newY) noexcept { y = newY; }

private:
    double x, y;
};

constexpr Point p1(9.4, 27.7);           // fine
constexpr Point p2(28.8, 5.3);         // also fine
```

Literal UDTs

Given

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
    return { (p1.xValue() + p2.xValue()) / 2, // call constexpr
            (p1.yValue() + p2.yValue()) / 2 }; // member funcs
}
```

a constexpr Point object could be defined as follows:

```
constexpr auto mid = midpoint(p1, p2); // init constexpr
// object w/result of
// constexpr function
```

Literal UDTs

In C++14, even setters can be constexpr:

```
class Point {
public:
    ...
    constexpr void setX(double newX) noexcept           // C++14 only
    { x = newX; }
    constexpr void setY(double newY) noexcept           // C++14 only
    { y = newY; }
    ...
};
```

Literal UDTs

Hence:

```
// return reflection of p with respect to the origin
constexpr Point reflection(const Point& p) noexcept
{
    Point result;                // create non-const Point
    result.setX(-p.xValue());    // set its x and y values
    result.setY(-p.yValue());
    return result;               // return constexpr copy of it
}

constexpr Point p1(9.4, 27.7);   // as before
constexpr Point p2(28.8, 5.3);

constexpr auto mid = midpoint(p1, p2); // as before
constexpr auto reflectedMid =       // reflectedMid's value
    reflection(mid);                 // is (-19.1 -16.5) and
                                     // a compile-time
                                     // constant
```

constexpr Pros and Cons

Pros:

- Can shift runtime work to compile time.
 - ➔ May help move objects from RAM to ROM.
- `constexpr` begets `constexpr`.
 - ➔ `constexpr` objects/functions facilitate creation of more `constexpr` objects/functions.

Con:

- `constexpr` is interface.
 - ➔ Removing `constexpr` can break arbitrary amounts of code.
 - ➔ Use `constexpr` only if you're willing to commit to it.

constexpr



Enhanced enums



Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.

Slide 33

Prefer Scoped enums to Unscoped enums

Unscoped (“old”) enums have several weaknesses:

```
enum Color { red, green, blue };
```

- **Namespace pollution:** red, green, blue in scope containing enum.

```
enum Color { red, green, blue };
void blue();                // error!
```

- **Unspecified size:** sizeof(Color) up to implementation.

- **Declaration-only not allowed:** definition must be given.

```
enum Options;                // error!
```

Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.

Slide 34

Prefer Scoped enums to Unscoped enums

- **Weak typing:** enumerators implicitly convert to int.

```
enum Color { red, green, blue };
Color c;
...
double d = c;           // Color→int→double
if (c) ...             // Color→int→bool
enum Options { debug, optimize, log };
Options opt;
...
if (c == opt) ...     // compare Color and Options!
```

Scoped enums

Scoped enums (“enum classes”) address all these issues:

```
enum class Color;           // fine
enum class Color { red, green, blue };
void blue();               // fine
Color c;
...
double d = c;             // error!
if (c) ...                // error!
enum class Options { debug, optimize, log };
Options opt;
...
if (c == opt) ...        // error!
```

Underlying Type

Default underlying type for scoped enums is `int`:

```
enum class Color { red, green, blue }; // in binary code,
                                       // red, green, blue
                                       // are ints
```

Can be overridden:

```
enum class Color: std::uint64_t // in binary code,
{ red, green, blue };          // red, green, blue
                               // are std::uint64_ts
```

Underlying Type

Can now be specified for unscoped enums, too:

```
enum Options: std::uint8_t { debug, optimize, log };
```

Such enums may also be forward-declared:

```
enum Options: std::uint8_t; // forward declaration
void doWork(std::function<void()> f, // fine
            Options opts);
```

But other issues remain:

- Namespace pollution
- Weak typing

enums, std::tuple, and std::get

Use of scoped enums with std::get verbose:

```
using StackOverflowData =
    std::tuple<std::string,           // name
              std::size_t, std::size_t, std::size_t, // badges
              std::size_t>;        // reputation

enum class SOIndices {
    name, bronzeCount, silverCount, goldCount, rep
};

void processSOUser(const StackOverflowData& user)
{
    auto badgeCount =
        std::get<std::size_t(SOIndices::bronzeCount)>(user) +
        std::get<std::size_t(SOIndices::silverCount)>(user) +
        std::get<std::size_t(SOIndices::goldCount)>(user);

    ...
}
```

Even worse with static_casts...

enums, std::tuple, and std::get

Enum→int conversion makes use of unscoped enums simpler:

```
enum {
    name, bronzeCount, silverCount, goldCount, rep // not
};                                                  // enum
                                                    // class

...

void processSOUser(const StackOverflowData& user)
{
    auto badgeCount = std::get<bronzeCount>(user) +
                      std::get<silverCount>(user) +
                      std::get<goldCount>(user);

    ...
}
```

Casting to Underlying Type

Verbosity of scoped enum approach can be reduced a bit:

```
// cast enumerator to underlying type
template<typename E>
constexpr typename std::underlying_type<E>::type
    toUType(E enumerator)
{
    return
        static_cast<typename
            std::underlying_type<E>::type>(enumerator);
}

void processSOUser(const StackOverflowData& user)
{
    auto badgeCount =
        std::get<toUType(SOIndices::bronzeCount)>(user) +
        std::get<toUType(SOIndices::silverCount)>(user) +
        std::get<toUType(SOIndices::goldCount)>(user);
    ...
}
```

toUType also more general than having clients cast to `std::size_t`:

- Not all enum-integral conversions call for `std::size_t`s.

Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013 Scott Meyers, all rights reserved.

Slide 41

Casting to Underlying Type

Alias template can hide “`template blah_blah_blah::type`” syntax:

```
template<typename E> // from previous slide
constexpr
typename std::underlying_type<E>::type toUType(E enumerator)
{
    return
        static_cast<typename
            std::underlying_type<E>::type>(enumerator);
}

template<typename E> // alias
using UnderlyingType = // template
    typename std::underlying_type<E>::type;

template<typename E> // revised
constexpr UnderlyingType<E> toUType(E enumerator) // toUType
{
    return static_cast<UnderlyingType<E>>(enumerator);
}
```

Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013 Scott Meyers, all rights reserved.

Slide 42

Casting to Underlying Type

C++14's auto return type eliminates type repetition.

- Without alias template:

```
template<typename E>
constexpr auto toUType(E enumerator)           // C++14
{
    return
        static_cast<typename
            std::underlying_type<E>::type>(enumerator);
}
```

- With alias template:

```
template<typename E>
using UnderlyingType = typename std::underlying_type<E>::type;

template<typename E>
constexpr auto toUType(E enumerator)           // C++14
{
    return static_cast<UnderlyingType<E>>(enumerator);
}
```

Enhanced enums



Further Information



Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.

Slide 45

Further Information

C++11 and C++14 in embedded systems:

- [“Embedded Programming with C++11,”](#) Rainer Grimm, *Meeting C++*, November 2013.
 - ➔ Overviews prevention of narrowing conversions, `static_assert`, user-defined literals, `constexpr`, generalized PODs, `std::array`, move semantics, perfect forwarding, and smart pointers.
- [“C++14 Adds Embedded Features,”](#) William Wong, *electronic design*, 19 August 2014.
 - ➔ Brief coverage of `auto` for lambda parameters and function return types, in-class aggregate initialization, user-defined literals, standard unit suffixes, `constexpr`, binary literals, and the digit separator.

Scott Meyers, Software Development Consultant
<http://aristeia.com/>

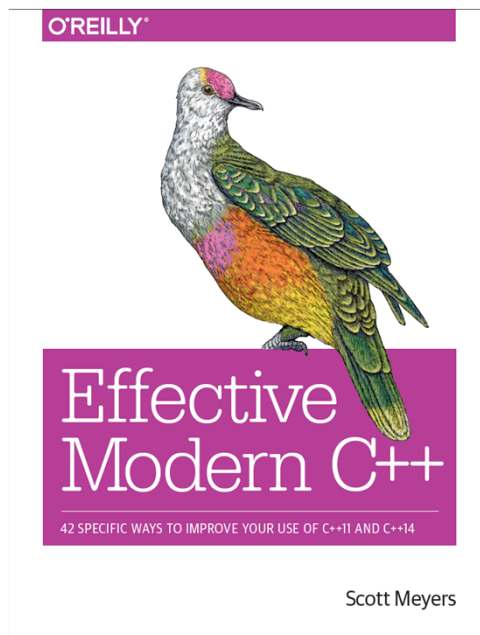
© 2013-14 Scott Meyers, all rights reserved.

Slide 46

Further Information

Effective use of C++11 and C++14:

- *Effective Modern C++*, Scott Meyers, O'Reilly, 2015.
 - ➔ Due out any day now...



Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.

Slide 47

Further Information

C++11:

- *Programming Language — C++*, Document 14882:2011, ISO/IEC, 2011-09-01.
 - ➔ ISO product page: <http://tinyurl.com/5soe87> (CHF238 ≙ ~\$245).
 - ➔ ANSI product page: <http://tinyurl.com/8m6vb5m> (\$30).
 - ➔ First post-Standard draft (N3337): <http://tinyurl.com/6wkboer> (free).
 - ◆ Essentially identical to the standard.
- *C++11*, Wikipedia.
- *Overview of the New C++ (C++11/14)*, Scott Meyers, http://www.artima.com/shop/overview_of_the_new_cpp.
 - ➔ Annotated training materials (analogous to these).
- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.



Scott Meyers, Software Development Consultant
<http://aristeia.com/>

© 2013-14 Scott Meyers, all rights reserved.

Slide 48

Further Information

C++14:

- [Working Draft, Standard for Programming Language C++](#), Document N3797, ISO/IEC, 2013-10-13.
 - ➔ Final freely available draft version of C++14.
- [C++14](#), Wikipedia.
- [“The view from C++ Standard meeting April 2013,”](#) Michael Wong, *C/C++ Cafe* (blog) :
 - ➔ Part 1, 25 April 2013. Discusses core language.
 - ➔ Part 2, 26 April 2013. Discusses standard library.
 - ➔ Part 3, 29 April 2013. Discusses concurrency and TSes.
- [Overview of the New C++ \(C++11/14\)](#), Scott Meyers, http://www.artima.com/shop/overview_of_the_new_cpp.
 - ➔ Annotated training materials (analogous to these).

Further Information

auto:

- [“C++11, VC++11, and Beyond,”](#) Herb Sutter, *Going Native 2012*, 3 February 2012.
 - ➔ Summarizes pros and cons of using auto.
- [“In what way can C++0x standard help you eliminate 64-bit errors,”](#) Andrey Karpov, viva64.com, 28 February 2010.
 - ➔ Source of “type shortcut” example.
- [“STL11: Magic && Secrets,”](#) Stephan T. Lavavej, *Going Native 2012*, 2 February 2012.
 - ➔ Source of accidental temporary example.
- [“Inferring too much,”](#) Motti Lanzkron, *I will not buy this blog, it is scratched!*, 21 February 2011.

Further Information

constexpr:

- [“constexpr – Generalized Constant Expressions in C++11,”](#) Alex Allain, [cprogramming.com](#).
- [“Using constexpr to Improve Security, Performance, and Encapsulation in C++,”](#) Danny Kalev, *Software Quality Matters Blog*, 19 December 2012.
- [“constexpr,”](#) Jarryd Beck, *The New C++* (blog), 14 November 2011.

Further Information

Enhanced enums:

- [“Strongly typed enumerations,”](#) *Wikipedia*, <http://tinyurl.com/7szjdey>.
 - ➔ Subsection of entry for “C++11”.
- [“enum class -- scoped and strongly typed enums,”](#) Bjarne Stroustrup, <http://tinyurl.com/2cvylcc>.
 - ➔ Subsection of “C++11 – the recently approved new ISO C++ standard.”
- [“Closer to Perfection: Get to Know C++11 Scoped and Based Enum Types,”](#) Danny Kalev, *smartbear.com*, 6 February 2013.
- [“Better types in C++11 - nullptr, enum classes \(strongly typed enumerations\) and cstdint,”](#) Alex Allain, [CProgramming.com](#).

Further Information

Alias templates:

- “Handling dependent names,” R. Martinho Fernandes, *Flaming Dangerzone*, 27 May 2012.

Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- **Commercial use:** <http://aristeia.com/Licensing/licensing.html>
- **Personal use:** <http://aristeia.com/Licensing/personalUse.html>

Courses currently available for personal use include:



About Scott Meyers



Scott offers training and consulting services on the design and implementation of C++ software systems. His web site,

<http://aristeia.com/>

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog