Some Lesser-Known Features of C++11 and C++14

Petr Zemek

Principal Developer
T&I, Threat Labs
https://petrzemek.net

C++ Developers' Guild, 2016-12-02

Reference Qualifiers

Distinguishing between `const` and non-`const`:

```
class Container {
public:
    iterator begin();
    const_iterator begin() const;
};
```

Reference Qualifiers

Distinguishing between `const` and non-`const`:

```cpp
class Container {
public:
    iterator begin();
    const_iterator begin() const;
};
```

How to distinguish between lvalue and rvalue?

Reference Qualifiers

Distinguishing between `const` and non-`const`:

```
class Container {
public:
    iterator begin();
    const_iterator begin() const;
};
```

How to distinguish between lvalue and rvalue?

```
A a;
a.foo();   // want lvalue version of foo()
A().foo(); // want rvalue version of foo()
```

Reference Qualifiers

Distinguishing between `const` and non-`const`:

```
class Container {
public:
    iterator begin();
    const_iterator begin() const;
};
```

How to distinguish between lvalue and rvalue?

```
A a;
a.foo();   // want lvalue version of foo()
A().foo(); // want rvalue version of foo()
```

C++11

```
class A {
public:
    void foo() &;  // for lvalues
    void foo() &&; // for rvalues
};
```

Reference Qualifiers (Continued)

Example (http://stackoverflow.com/a/8614126/2580955):

```
class S {
public:
    S& operator ++();
    S* operator &();
};
```

Reference Qualifiers (Continued)

Example (http://stackoverflow.com/a/8614126/2580955):

```
class S {
public:
    S& operator ++();
    S* operator &();
};

S() = S(); // rvalue as lhs of assignment
```

Reference Qualifiers (Continued)

Example (http://stackoverflow.com/a/8614126/2580955):

```
class S {
public:
    S& operator ++();
    S* operator &();
};

S() = S(); // rvalue as lhs of assignment
++S();     // incrementing rvalue
```

Reference Qualifiers (Continued)

Example (http://stackoverflow.com/a/8614126/2580955):

```
class S {
public:
    S& operator ++();
    S* operator &();
};

S() = S(); // rvalue as lhs of assignment
++S();     // incrementing rvalue
&S();      // taking address of rvalue
```

Reference Qualifiers (Continued)

Example (http://stackoverflow.com/a/8614126/2580955):

```
class S {
public:
      S& operator ++();
      S* operator &();
};

S() = S(); // rvalue as lhs of assignment
++S();     // incrementing rvalue
&S();      // taking address of rvalue
```

The fix is simple:

```
class S {
public:
      S& operator ++() &;
      S* operator &() &;
      const S& operator =(const S&) &;
};
```

Inline Namespaces

Where you may encounter `inline`:

- inline functions (C99, C++98)

Inline Namespaces

Where you may encounter `inline`:

- inline functions (C99, C++98)
- inline namespaces (C++11)

Inline Namespaces

Where you may encounter `inline`:

- inline functions (C99, C++98)
- inline namespaces (C++11)
- inline variables (C++1z)

Inline Namespaces

Where you may encounter `inline`:

- inline functions (C99, C++98)
- inline namespaces (C++11)
- inline variables (C++1z)

C++11

```
namespace A {
    inline namespace B {
        class C {};
    }
}
```

Inline Namespaces

Where you may encounter `inline`:

- inline functions (C99, C++98)
- inline namespaces (C++11)
- inline variables (C++1z)

C++11

```
namespace A {
    inline namespace B {
        class C {};
    }
}

A::B::C c1; // OK (expected)
```

Inline Namespaces

Where you may encounter `inline`:

- inline functions (C99, C++98)
- inline namespaces (C++11)
- inline variables (C++1z)

C++11

```
namespace A {
    inline namespace B {
        class C {};
    }
}

A::B::C c1; // OK (expected)
A::C c2;    // also OK because B is inline
```

Inline Namespaces (Continued)

Raison d'être: library versioning

Inline Namespaces (Continued)

Raison d'être: library versioning

```
namespace lib {
    namespace v1 {
        class MyClass { /* ... */ };
    }
    inline namespace v2 {
        class MyClass { /* ... */ };
    }
}
```

Inline Namespaces (Continued)

Raison d'être: library versioning

```
namespace lib {
    namespace v1 {
        class MyClass { /* ... */ };
    }
    inline namespace v2 {
        class MyClass { /* ... */ };
    }
}
```

Where you may have seen them in the standard library:

```
using namespace std::literals::string_literals;
using namespace std::string_literals;
using namespace std::literals;
```

Inheriting Constructors

```
class A {
public:
    A(int i, int j): i(i), j(j) {}

private:
    int i, j;
};
```

Inheriting Constructors

```
class A {
public:
    A(int i, int j): i(i), j(j) {}

private:
    int i, j;
};
```

C++98

```
class B: public A {
    B(int i, int j): A(i, j) {}
};
```

```
class A {
public:
    A(int i, int j): i(i), j(j) {}

private:
    int i, j;
};
```

C++98

```
class B: public A {
    B(int i, int j): A(i, j) {}
};
```

C++11

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

Delegating Constructors

C++11

```
class A {
public:
    A(): A(42) {} // delegation
    A(int i): i(i) {}

private:
    int i;
};
```

Delegating Constructors

C++11

```
class A {
public:
    A(): A(42) {} // delegation
    A(int i): i(i) {}

private:
    int i;
};
```

What is a downside of this C++98 alternative?

```
class A {
public:
    A(int i = 42): i(i) {}

private:
    int i;
};
```

Delegating Constructors (Continued)

Beware:

```cpp
class A {
public:
    A() { A(42); } // oops...
    A(int i): i(i) {}

private:
    int i;
};
```

Explicit Conversion Operators

Motivation:

```
SmartPtr<int> p;
if (p) { // ...
```

Explicit Conversion Operators

Motivation:

```cpp
SmartPtr<int> p;
if (p) { // ...
```

Implementation?

```cpp
template<typename T>
class SmartPtr {
public:
    operator bool() const;

    // ...
};
```

Explicit Conversion Operators

Motivation:

```
SmartPtr<int> p;
if (p) { // ...
```

Implementation?

```
template<typename T>
class SmartPtr {
public:
    operator bool() const;

    // ...
};
```

Oh oh...

```
SmartPtr<Car> p1;
SmartPtr<Person> p2; // Person is unrelated to Car
if (p1 == p2) {      // OK... (?!)
```

Explicit Conversion Operators (Continued)

How would you solve this in C++98?

Explicit Conversion Operators (Continued)

How would you solve this in C++98?

C++11

```
template<typename T>
class SmartPtr {
public:
    explicit operator bool() const;

    // ...
};
```

Explicit Conversion Operators (Continued)

How would you solve this in C++98?

C++11

```cpp
template<typename T>
class SmartPtr {
public:
    explicit operator bool() const;

    // ...
};
```

```cpp
if (p) {          // OK
if (p1 == p2) { // fails to compile
```

Speaking of explicit...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();
```

Speaking of `explicit`...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                           // no
    explicit A(int i);
```

Speaking of explicit...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                                  // no
    explicit A(int i);                             // yes
    explicit A(int i = 1);
```

Speaking of `explicit`...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                               // no
    explicit A(int i);                          // yes
    explicit A(int i = 1);                      // yes
    explicit A(int i = 1, int j = 2);
```

Speaking of explicit...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                          // no
    explicit A(int i);                     // yes
    explicit A(int i = 1);                 // yes
    explicit A(int i = 1, int j = 2); // yes
    explicit A(int i, int j);
```

Speaking of `explicit`...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                          // no
    explicit A(int i);                     // yes
    explicit A(int i = 1);                 // yes
    explicit A(int i = 1, int j = 2); // yes
    explicit A(int i, int j);              // no
    explicit A(const A& other);
```

Speaking of `explicit`...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                          // no
    explicit A(int i);                     // yes
    explicit A(int i = 1);                 // yes
    explicit A(int i = 1, int j = 2); // yes
    explicit A(int i, int j);              // no
    explicit A(const A& other);            // yes
```

Speaking of `explicit`...

Does `explicit` have any effect here (C++98)?

```
class A {
    explicit A();                         // no
    explicit A(int i);                    // yes
    explicit A(int i = 1);                // yes
    explicit A(int i = 1, int j = 2);     // yes
    explicit A(int i, int j);             // no
    explicit A(const A& other);           // yes
};
```

C++11

```
class A {
public:
    explicit A(int i, int j);
};

A a{1, 2};      // OK
A b = {1, 2};   // fails to compile
```

New String Literals

C++11

```
const char     *s1 = u8"UTF-8 string literal";
const char16_t *s2 = u"UTF-16 string literal";
const char32_t *s3 = U"UTF-32 string literal";
```

New String Literals

C++11

```
const char     *s1 = u8"UTF-8 string literal";
const char16_t *s2 = u"UTF-16 string literal";
const char32_t *s3 = U"UTF-32 string literal";
```

Inserting Unicode codepoints: \uNNNN and \UNNNNNNNN

New String Literals

C++11

```
const char     *s1 = u8"UTF-8 string literal";
const char16_t *s2 = u"UTF-16 string literal";
const char32_t *s3 = U"UTF-32 string literal";
```

Inserting Unicode codepoints: \uNNNN and \UNNNNNNNN

C++11

```
std::regex pattern(R"(\d{1,3}:"[a-d]")");

std::string code(R"(
    int main() {
        return 0;
    }
)");
```

Binary Literals

```
auto a = 42;   // decimal
auto b = 0x2a; // hexadecimal
auto c = 052;  // octal
```

Binary Literals

```
auto a = 42;    // decimal
auto b = 0x2a; // hexadecimal
auto c = 052;  // octal
```

C++14

```
auto d = 0b101010; // 42
```

Digit Separators

```
const auto AREA = 1243245845;
```

Digit Separators

```cpp
const auto AREA = 1243245845;
```

C++14

```cpp
const auto AREA = 1'243'245'845;
```

Digit Separators

```cpp
const auto AREA = 1243245845;
```

C++14

```cpp
const auto AREA = 1'243'245'845;
const auto MASK = 0b1000'0001'1000'0000;
```

User-Defined Literals

C++11

```
std::string operator "" _s(const char *str,
        std::size_t length) {
    return std::string(str, length);
}

auto name = "Petr Zemek"_s;
```

User-Defined Literals

C++11

```
std::string operator "" _s(const char *str,
            std::size_t length) {
      return std::string(str, length);
}

auto name = "Petr Zemek"_s;

std::string s1 = "abc\x00xyz";    // s1: "abc"
std::string s2 = "abc\x00xyz"_s; // s2: "abc\x00xyz"
```

User-Defined Literals

C++11

```
std::string operator "" _s(const char *str,
        std::size_t length) {
    return std::string(str, length);
}

auto name = "Petr Zemek"_s;
```

```
std::string s1 = "abc\x00xyz";   // s1: "abc"
std::string s2 = "abc\x00xyz"_s; // s2: "abc\x00xyz"
```

Note: naming convention

Standard User-Defined Literals

C++14

- `s` (`std::string` literals)
- `if`, `i`, `il` (`std::complex` literals)
- `h`, `min`, `s`, `ms`, `us`, `ns` (`std::chrono::duration` literals)

Standard User-Defined Literals

C++14

- `s` (`std::string` literals)
- `if`, `i`, `il` (`std::complex` literals)
- `h`, `min`, `s`, `ms`, `us`, `ns` (`std::chrono::duration` literals)

Example:

```
using namespace std::literals;

auto name = "Petr Zemek"s;
auto runtime = 30s;
```

Alternative Function Syntax

C++11

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

Alternative Function Syntax

C++11

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

- use in generic code

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

Alternative Function Syntax

C++11

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

- use in generic code

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

- eliminates repetition

```
class LongClassName {
    using IntVec = std::vector<int>;
    IntVec f();
};

auto LongClassName::f() -> IntVec {/*..*/}
// vs
LongClassName::IntVec LongClassName::f() {/*..*/}
```

Alternative Function Syntax (Continued)

- may lead to more readable code

```
void (*get_func_on(int i))(int);
```

Alternative Function Syntax (Continued)

- may lead to more readable code

```
void (*get_func_on(int i))(int);
// vs
auto get_func_on(int i) -> void (*)(int);
```

Alternative Function Syntax (Continued)

- may lead to more readable code

```
void (*get_func_on(int i))(int);
// vs
auto get_func_on(int i) -> void (*)(int);
```

- (H. Sutter) the C++ world is moving to a left-to-right declaration style everywhere:

  category name = **type** and/or initializer ;

  where category is either `auto` or `using`

Extern Templates

```
// module1.cpp
std::vector<MyClass> v;

// module2.cpp
std::vector<MyClass> w;
```

Extern Templates

```cpp
// module1.cpp
std::vector<MyClass> v;

// module2.cpp
std::vector<MyClass> w;
```

C++11

```cpp
// module1.cpp
extern template class std::vector<MyClass>;
// ...

// module2.cpp
template class std::vector<MyClass>;
// ...
```

Extern Templates

```
// module1.cpp
std::vector<MyClass> v;

// module2.cpp
std::vector<MyClass> w;
```

C++11

```
// module1.cpp
extern template class std::vector<MyClass>;
// ...

// module2.cpp
template class std::vector<MyClass>;
// ...
```

Note: not to be confused with exported templates (`export`)

TransformationTraits Redux

C++11

```
typename std::remove_reference<T>::type
```

TransformationTraits Redux

C++11

```cpp
typename std::remove_reference<T>::type
```

C++14

```cpp
std::remove_reference_t<T>
```

TransformationTraits Redux

C++11

```
typename std::remove_reference<T>::type
```

C++14

```
std::remove_reference_t<T>
```

Implementation:

```
template<typename T>
using remove_reference_t = \
    typename remove_reference<T>::type;
```

Variable Templates

C++14

```
template<typename T>
constexpr T pi = T(3.14159265358979323846);
```

C++14

```
template<typename T>
constexpr T pi = T(3.1415926535897932384 6);

template<typename T>
T area_of_circle_with_radius(T r) {
    return pi<T> * r * r;
}
```

Variable Templates (Continued)

C++11

```
template<typename T>
struct is_lvalue_reference; // in namespace std
```

Variable Templates (Continued)

C++11

```
template<typename T>
struct is_lvalue_reference; // in namespace std

std::is_lvalue_reference<int>::value  // false
std::is_lvalue_reference<int&>::value // true
```

Variable Templates (Continued)

C++11

```
template<typename T>
struct is_lvalue_reference; // in namespace std

std::is_lvalue_reference<int>::value  // false
std::is_lvalue_reference<int&>::value // true
```

C++1z

```
std::is_lvalue_reference_v<int>  // false
std::is_lvalue_reference_v<int&> // true
```

Final Classes

C++98

```cpp
class NotBase {}; // Do not subclass this class!

class Derived: public NotBase {};
```

Final Classes

C++98

```cpp
class NotBase {}; // Do not subclass this class!

class Derived: public NotBase {};
```

Can this be enforced in C++98?

Final Classes

C++98

```
class NotBase {}; // Do not subclass this class!

class Derived: public NotBase {};
```

Can this be enforced in C++98?

C++11

```
class NotBase final {};

class Derived: public NotBase {}; // comp. error
```

Final Classes

C++98

```
class NotBase {}; // Do not subclass this class!

class Derived: public NotBase {};
```

Can this be enforced in C++98?

C++11

```
class NotBase final {};

class Derived: public NotBase {}; // comp. error
```

Note: `final` is not a reserved word

Final Virtual Member Functions

C++98

```
class Base {
    virtual void f(); // Please, do not override it.
};
class Derived: public Base {
    virtual void f();
};
```

Final Virtual Member Functions

C++98

```cpp
class Base {
    virtual void f(); // Please, do not override it.
};
class Derived: public Base {
    virtual void f();
};
```

Can this be enforced in C++98?

Final Virtual Member Functions

C++98

```
class Base {
    virtual void f(); // Please, do not override it.
};
class Derived: public Base {
    virtual void f();
};
```

Can this be enforced in C++98?

C++11

```
class Base {
    virtual void f() final;
};
class Derived: public Base {
    virtual void f(); // compilation error
};
```

Standardized Attribute Syntax

C++11

```
[[atribute_name(parameters)]]
```

Standardized Attribute Syntax

C++11

`[[atribute_name(parameters)]]`

Standard attributes:

- `[[noreturn]]` (C++11)

```
[[noreturn]]
void my_exit(int code);
```

Standardized Attribute Syntax

C++11

`[[atribute_name(parameters)]]`

Standard attributes:

- `[[noreturn]]` (C++11)

```
[[noreturn]]
void my_exit(int code);
```

- `[[deprecated]]` (C++14)

```
[[deprecated("use bar()")]]
void foo();

foo() // warning: 'foo' is deprecated: use bar()
```

Standardized Attribute Syntax

C++11

`[[atribute_name(parameters)]]`

Standard attributes:

- `[[noreturn]]` (C++11)

```cpp
[[noreturn]]
void my_exit(int code);
```

- `[[deprecated]]` (C++14)

```cpp
[[deprecated("use bar()")]]
void foo();

foo() // warning: 'foo' is deprecated: use bar()
```

- `[[carries_dependency]]` (C++11)

A New Specifier: `noexcept`

C++11

```
void foo() noexcept;
```

A New Specifier: `noexcept`

C++11

```
void foo() noexcept;
```

Notes:

- similar to `throw()` in C++98

A New Specifier: `noexcept`

C++11

```
void foo() noexcept;
```

Notes:

- similar to `throw()` in C++98
- `std::terminate()`

A New Specifier: `noexcept`

C++11

```
void foo() noexcept;
```

Notes:

- similar to `throw()` in C++98
- `std::terminate()`
- `noexcept(expr)`

A New Specifier: `noexcept`

C++11

```
void foo() noexcept;
```

Notes:

- similar to `throw()` in C++98
- `std::terminate()`
- `noexcept(expr)`
- destructors are `noexcept` by default

A New Specifier: `noexcept`

C++11

```
void foo() noexcept;
```

Notes:

- similar to `throw()` in C++98
- `std::terminate()`
- `noexcept(expr)`
- destructors are `noexcept` by default
- motivation: move semantics

Forward Declaration of Enumerators

```
enum A; // compilation error (missing base type)
```

Forward Declaration of Enumerators

```cpp
enum A; // compilation error (missing base type)
```

C++11

```cpp
enum B : short;         // OK
```

Forward Declaration of Enumerators

```
enum A; // compilation error (missing base type)
```

C++11

```
enum B : short;        // OK
enum class D : short; // OK
```

Forward Declaration of Enumerators

```cpp
enum A; // compilation error (missing base type)
```

C++11

```cpp
enum B : short;         // OK
enum class D : short; // OK
enum class E;           // OK (base type is int)
```

Improved Compatibility with C99

- `long long int`

Improved Compatibility with C99

- `long long int`
- `__func__` (identifier)

Improved Compatibility with C99

- `long long int`
- `__func__` (identifier)
- standard header files (e.g. `<cstdint>`)

Improved Compatibility with C99

- `long long int`
- `__func__` (identifier)
- standard header files (e.g. `<cstdint>`)
- variadic macros

Improved Compatibility with C99

- `long long int`
- `__func__` (identifier)
- standard header files (e.g. `<cstdint>`)
- variadic macros

```
void _dgbprintf(const char *file, int line,
                const char *fmt, ...);

#define dbgprintf(...) \
    _dbgprintf(__FILE__, __LINE__, __VA_ARGS__)
```

Removals/Deprecations

Removals:

- meaning of the `export` keyword

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`
- conversion from string literals to `char *`

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`
- conversion from string literals to `char *`

Deprecations:

- increment of `bool` with `++`

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`
- conversion from string literals to `char *`

Deprecations:

- increment of `bool` with `++`
- meaning of the `register` keyword

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`
- conversion from string literals to `char *`

Deprecations:

- increment of `bool` with `++`
- meaning of the `register` keyword
- dynamic exception specifications, `std::unexpected()`

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`
- conversion from string literals to `char *`

Deprecations:

- increment of `bool` with `++`
- meaning of the `register` keyword
- dynamic exception specifications, `std::unexpected()`
- `std::auto_ptr`

Removals/Deprecations

Removals:

- meaning of the `export` keyword
- access declarations (pre-C++98 syntax)
- `std::gets()`
- conversion from string literals to `char *`

Deprecations:

- increment of `bool` with `++`
- meaning of the `register` keyword
- dynamic exception specifications, `std::unexpected()`
- `std::auto_ptr`
- several other features (function object base classes, adapters, binders)

Shameless Plug

Petr Zemek
Co je nového v C++11
https://cs-blog.petrzemek.net/2012-12-04-co-je-noveho-v-cpp11

Petr Zemek
Co je nového v C++14
https://cs-blog.petrzemek.net/2014-09-20-co-je-noveho-v-cpp14

Petr Zemek
Méně známé novinky v C++11 a C++14
https://cs-blog.petrzemek.net/2015-10-06-mene-zname-novinky-v-cpp11-a-cpp14

Petr Zemek
Improving C++98 Code With C++11
https://blog.petrzemek.net/2014/12/07/improving-cpp98-code-with-cpp11/

Bonus

Member Initializers and Aggregates

C++11

```
struct A {
    int i = 0;
    double j = 0.0;
};
```

Member Initializers and Aggregates

C++11

```
struct A {
    int i = 0;
    double j = 0.0;
};
```

In C++11, watch out:

```
A a = {1};        // OK (C++14), comp. error in C++11
A b = {1, 2.0}; // OK (C++14), comp. error in C++11
```

Member Initializers and Aggregates

C++11

```
struct A {
    int i = 0;
    double j = 0.0;
};
```

In C++11, watch out:

```
A a = {1};      // OK (C++14), comp. error in C++11
A b = {1, 2.0}; // OK (C++14), comp. error in C++11
```

Note: `gcc-4.9 -std=c++14` also fails with a compilation error

Extended Friend Declarations

C++11

```
class A;
class B {
    friend class A; // OK (old style)
    friend A;       // OK since C++11
};
```

Extended Friend Declarations

C++11

```cpp
class A;
class B {
    friend class A; // OK (old style)
    friend A;       // OK since C++11
};
```

You can now declare template parameters as friends:

```cpp
template<typename T>
class C {
    friend class T; // compilation error
    friend T;       // OK since C++11
};
```

Alignment Operators

C++11

```
using cacheline alignas(128) = char[128];

std::cout << alignof(cacheline) << '\n';
```

Unnamed/Local Types as Template Arguments

C++11

```
template<typename T>
void func(T t) {}

enum { e }; // unnamed type

int main() {
    struct A { int i; }; // local type
    A a;

    func(e); // OK in C++11, comp. error in C++98
    func(a); // OK in C++11, comp. error in C++98
}
```

`sizeof` Works On Non-Static Data Members

```
struct A {
    int i;
};
```

sizeof Works On Non-Static Data Members

```
struct A {
    int i;
};
```

C++98

```
sizeof A::i // compilation error
```

sizeof Works On Non-Static Data Members

```
struct A {
    int i;
};
```

C++98

```
sizeof A::i // compilation error
```

```
A a;
sizeof a.i  // OK
```

`sizeof` Works On Non-Static Data Members

```
struct A {
    int i;
};
```

C++98

```
sizeof A::i // compilation error

A a;
sizeof a.i  // OK
```

C++11

```
sizeof A::i // OK
```