Embracing Modern C++ *Safely*

By **Rostislav Khlebnikov** and **John Lakos**

Revised March 29th, 2018

Abstract

Modern C++, inaugurated by C++11, introduces many useful features. With an eye toward their successful adoption into large, mature code bases, we have cataloged useful new language features, along with any concomitant pitfalls, into three subcategories: (1) safe features, which can safely be used liberally wherever applicable, (2) conditionally-safe features, which can be used safely only under certain specific conditions, and (3) "unsafe" features, whose appropriate and safe use in production code is comparatively rare. Pitfalls associated with each feature in the second and third categories are elucidated objectively without endorsing any particular solution. Specific design decisions are, instead, left for the reader to determine in the context of his or her own particular development environment.

Introduction

In the years since C++11 became widely accessible, it has become well known that it introduced several additional pitfalls, which experienced software developers have since learned to work around. Although useful coding standards, design rules, and best practices are readily available[1], they vary widely in their scope, accuracy, and applicability. Moreover, the solutions, recommendations, and opinions rendered in books, online, and in individual companies' style guides are often subjective, occasionally controversial, and invariably different (sometimes incompatibly so) from one to the next.

In response to a clear and pressing need to better understand such important issues, this paper aims to provide carefully curated, thoroughly vetted facts and objectively verifiable observations elucidating just the specific nature of these new pitfalls – irrespective of any particular solution. We begin by identifying several new C++11 features that practical experience in large code bases has shown can safely be used liberally wherever applicable. We then consider some other new features that, while potentially valuable, nonetheless merit considerable discussion as to what constitutes their productive use. Finally, we address a couple of features whose practical use is limited to very specific and rare circumstances. In each case, the discussion of a feature along with any identified concerns is intended to focus and facilitate its use, rather than discourage it.

Safe features

Several of the C++11 language features simply standardize existing practice or address previous language deficiencies, are not prone to misuse, and can therefore be (safely) applied liberally throughout the code base:

- deleted and defaulted[2] functions
- alias templates
- strongly-typed enumerations (`enum class`)
- explicit conversion operators

[1] E.g., C++ Core Guidelines (see http://isocpp.github.io/CppCoreGuidelines).

[2] Note that an explicitly declared special member function – even when defaulted in class scope – will disable generation of other such functions in the same way as would a non-defaulted one. Also note that `default` preserves triviality *only* when applied within class scope.

- `override`
- local and unnamed types as template parameters
- compile-time assertions (`static_assert`)
- `decltype`[3]
- delegating and inheriting constructors
- null pointer literal (`nullptr`)
- Dynamic initialization and destruction with concurrency
- `extern template`
- raw and Unicode[4] string literals
- alignment control (`alignof` and `alignas`)
- attributes[5]
- inline namespaces
- trailing function return types
- generalized unions and PODs
- `long long`
- extended friend declarations
- consecutive right angle brackets in templates

These features, used appropriately, typically yield clearer, more concise code compared to what was achievable with older C++ standards.

Conditionally Safe Features

There are some new features in C++11 that, though often beneficial, can also lead to losses in performance, maintainability, and correctness. In this section we address pitfalls associated with several such C++11 constructs.

Type Deduction from Initializer (`auto`)

Use of the `auto` type placeholder obviates explicit specification of a variable's type, relying instead on the compiler to deduce the type from its initializer. As of C++17, however, it is not possible to place (compiler-enforceable) constraints on the `auto` portion of the deduced type, which (similarly to unconstrained templates) can lead to difficult-to-diagnose compile-time (and runtime) errors. Furthermore, lack of readily observable type information might impede human comprehension.

There are, however, situations in which `auto` can be used without compromising maintainability[6]:

- The variable is initialized via a factory function template such that the return type is derived from only explicitly provided type arguments:
```
auto ptr = std::make_shared<MyType>();  // Deduces 'std::shared_ptr<MyType>'.
```
- The deduced type is somehow clearly duplicated in the initializing expression:
```
const auto& value = std::get<MyType>(myVariant);  // Deduces 'MyType'.
```
- The variable's type is impossible to spell explicitly (e.g., lambdas[7]):

[3] Note that `decltype(x)` and `decltype((x))` do not necessarily represent the same type: The former interprets its argument as a *variable*; the latter, as an *expression*.

[4] Note that lack of distinct type for UTF-8-encoded character and string literals might lead to portability issues (see http://wg21.link/p0482r1).

[5] Note that the `[[carries_dependency]]` attribute pertains to use of `memory_order_consume`, which is actively discouraged as of the C++17 standard (see [atomics.order]).

[6] `auto` can also be used productively in range-based for loops (see below).

```
auto filterFn = [&](const MyType& value) { /* Perform the check. */ };
```

- The (precise) type of the variable is complex and does not communicate useful information (e.g., expression templates, range adapters):

```
auto sumExpr = matrix1 + matrix2;
auto resultExpr = sumExpr * matrix3 - 2 * matrix4;
  // Deduces 'Sub<Mul<Add<Matrix, Matrix>, Matrix>, Mul<Terminal, Matrix>>'.

auto resultRange = rng | filtered(FiltFn()) | transformed(TransformFn());
  // Deduces 'TransformRng<FilterRng<decltype(rng), FiltFn>, TransformFn>'.
```

`auto` can also be used to inform the reader that a function does not rely on the specific type of a variable, but instead on a *concept*[8] – i.e., a named set of type constraints – that the variable's type is supposed to satisfy. Ideally, this concept would be (1) obvious from the initializing expression, and (2) a *vocabulary concept* (invariably known to the reader). Currently, the only set of concepts (possibly) satisfying both of these conditions for virtually all experienced C++ developers is the *Iterator* concept hierarchy:

```
auto it = container.begin();
  // Type 'decltype(it)' satisfies at least the 'InputIterator' concept.
```

Lack of ability to specify a concept that the `auto`-deduced type satisfies, however, might lead to failure to express essential semantic information – e.g., when subsequent (perhaps even yet-to-be-written) code requires operations beyond what explicit use of an `InputIterator` concept would have required.

Appropriate reference and cv-qualifier specifications for `auto` are, of course, nonetheless necessary to prevent unintended copies:

```
auto value = std::get<MyType>(myVariant);
  // 'value' is a copy of the data already stored in 'myVariant'.
  // Attempts to change 'myVariant' through 'value' will compile, but will not work.
```

Even a fully cv-ref-qualified `auto` might still prove inadequate in cases as simple as introducing a variable for a returned-temporary value:

```
// refactoring useValue(getValue()):
auto&& tempValue = getValue();
useValue(tempValue);
  // WARNING - Might change semantics! Equivalent code must also use 'std::forward':
  //   useValue(std::forward<decltype(tempValue)>(tempValue));
```

When converting to a particular type, compulsive use of auto might also lead to employing less-safe (e.g., lossy) *explicit* conversions, rather than potentially more-safe (e.g., non-lossy) *implicit* ones (perhaps deliberately provided as a consequence of thoughtful library design):

```
template <class Duration1, class Duration2>
std::chrono::seconds combine_durations(Duration1 d1, Duration2 d2) {
  // auto d = std::chrono::seconds{d1 + d2};
  // BAD - Invokes explicit conversion: Would compile and run for two integers!

  std::chrono::seconds d = d1 + d2; // BETTER - Would NOT compile for two integers.
}
```

Similarly, explicit conversions encouraged by such unbridled use of `auto` might also lead to maintenance issues:

```
void func(Derived *pDerived) { // Refactored from std::shared_ptr<Derived>
  auto pBase = std::shared_ptr<Base>{pDerived};
    // BAD - 'pBase' now assumes duplicate ownerships (resulting in UB).

  std::shared_ptr<Base> pBase = pDerived; // OK - now fails to compile.
}
```

[7] Unlike using `auto` to deduce lambda types, employing polymorphic function wrappers, such as `std::function`, might incur runtime overhead due to potential memory allocation upon construction, as well as virtual dispatch on every call.

[8] Note that, as of C++17, *concepts* as a *language feature* were not yet a part of the C++ Standard.

Finally, using `auto` for deducing fundamental types (e.g., `int`, `float`) might hide important, context-sensitive considerations, such as overflow and mixing `signed` and `unsigned` integral types, which depend critically on the number and precise interpretation of the respective bits.

Lambda Expressions

Lambda expressions provide a lightweight syntax for introducing unnamed function objects (closures), often enabling easier-to-understand, more maintainable code. Using lambdas instead of existing (or appropriately extracted) named functions or function objects can, however, lead to excessive code duplication:

```
std::sort(v.begin(), v.end(), [](int lhs, int rhs) { return lhs > rhs; });
  // Using 'std::greater<int>()' provides both better expressiveness and modularity.
```

Lambdas that might outlive the scope in which they were introduced are susceptible to inadvertent dangling references or pointers – especially if reference *capture-default* (`[&]`) is used, or if the `this` pointer is captured (either implicitly or explicitly):

```
class MyHandler { EventType d_myEvent; /*...*/ }
void MyHandler::addTo(EventManager& em) {
    em.addHandler([=](EventType event) {
        if (d_myEvent == event) /* ... */
    });  // ^~~~~~~~~ The 'this' pointer is captured implicitly, and becomes
}        //           dangling when this object (of class 'MyHandler') is destroyed.
```

Range-based for Loops

Range-based `for` loops provide a more abstract, concise, and error-resistant iteration construct compared to traditional ones (i.e., those exposing an iterator or index). These newer `for` loops are, however, susceptible to being pressed into service where a standard algorithm (perhaps employing a lambda) might have provided a simpler, more expressive, and more maintainable solution.

Improper specification of the type of the `for`'s element variable might lead to an unnecessary (potentially sliced) copy of *each* element in a container:

```
for (std::string s : vectorOfStrings)
  // Makes a copy of each element in 'vectorOfStrings' - use 'const std::string&'.
struct Derived : Base { /* ... */ }
for (Base s : vectorOfDerived)
  // Makes a sliced copy of each element in 'vectorOfDerived' - use 'const Base&'.
```

Range-based `for` loops over associative containers, such as `map` and `unordered_map`, are especially prone to unintended copies, as the essential `const`ness of the *key* in their respective `value_type`s can be easy to overlook:

```
for (const std::pair<int, std::string>& keyVal : mapIntToString)
  // BAD - Makes a copy of each key-value pair.
for (const std::pair<const int, std::string>& keyVal : mapIntToString)
  // OK - No copy made (provided that corresponding types are maintained properly).
for (const std::map<int, std::string>::value_type& keyVal : mapIntToString)
  // OK - No copy made (provided that corresponding types are maintained properly).
for (const auto& keyVal : mapIntToString)
for (auto&& keyVal : mapIntToString)
  // OK - No copy made.  (The type of 'keyVal' is, however, elided completely.)
```

Finally, range-based for loops might hide issues with iterator invalidation and reference lifetime extension, leading to undefined behavior[9]:

```
std::optional<std::vector<int>> get_vector();
for (int value : get_vector().value()) { }
  // BAD - The object returned by 'get_vector' is destroyed before first iteration.
```

[9] For a more detailed discussion of this issue, see, e.g., https://abseil.io/tips/107.

`initializer_list`

The `initializer_list` type enables the creation of functions that operate on sequences of different (compile-time) size without having to templatize them. Employing an `initializer_list` might incur runtime overhead, as its construction behaves as if the compiler generated and materialized a temporary array, wherein elements are *copy-initialized* from the original sequence of initializers:

```
std::vector<std::string> vec = {str1, str2, str3};  // Copies input strings twice.
```

Furthermore, elements within an `initializer_list` cannot be *moved from*, making the `initializer_list` type unusable for move-only types.

BRACED INITIALIZATION

Braced initialization syntax facilitates initialization of containers with their contents, provides a convenient way to initialize simple classes having only public data members, prevents narrowing conversions, and avoids the "most vexing parse". Braced initialization, however, also implies that the `initializer_list` constructor is preferred even when another constructor would otherwise be an equivalent or better match for the arguments, which can lead to ***silent*** behavioral changes for existing clients if an `initializer_list` constructor is retrofitted into library code:

```
struct IntVector {
    IntVector(size_t count, int value);
    // IntVector(std::initializer_list<int> values);  // Might affect clients.
};

IntVector data{3, 1};
  // Uncommenting IntVector's initializer_list constructor
  // silently changes the contents of 'data' from { 1, 1, 1 } to { 3, 1 }.
```

Furthermore, this preference for `initializer_list` constructors might also lead to unexpected behavior resulting from implicit conversions:

```
std::string s{65, 'a'};  // Variable 's' has the value "Aa".
```

Finally, use of braced syntax for containers often tends to suggest that sequential element initialization will be performed, yet such is not necessarily the case – even within the standard library! For example, `std::vector<T>{3}` will have one or three elements, depending (respectively) on whether or not `T` is implicitly constructible from `3`: E.g., `std::vector<int>{3}` would contain only a single element (the integer `3`), whereas `std::vector<std::string>{3}` would contain three empty strings. Hence, such purely conventional expectations for the meaning of braced initialization can be especially problematic in generic code.

RVALUE REFERENCES, FORWARDING REFERENCES, AND MOVE SEMANTICS

Rvalue references enable support for *move-only* types as well as the differentiation of temporary values, thereby enabling optimizations resulting from the avoidance of superfluous copies. Throwing move operations – i.e., those not explicitly marked `noexcept` (or so generated by the compiler) – might, however, impede such optimizations. For example, operations defined for `std::vector` providing *the strong exception-safety guarantee* (e.g., `push_back`) will be forced to fall back on copying and even on providing ***only*** the *basic* guarantee, if the elements are not copyable:

```
struct MyType {
    std::unique_ptr<int> d_ptr;
    MyCxx03TypeWithThrowingCopy d_value;
};
```

```
  // 'MyType' is non-copyable (due to its 'd_ptr' member) and its (compiler-generated)
  // move assignment is 'noexcept(false)' (due to the 'd_value' member); hence,
  // 'vector<MyType>::push_back' is NOT able to ensure the strong guarantee!
```
A named variable of rvalue-reference type (such as a *move* constructor's parameter) is itself an lvalue; hence, explicit casts (usually performed using `std::move`) are required to suppress unwanted copies – both when moving the object as a whole and when moving its sub-objects:
```
  void func(std::vector<int>&& initData) {
      // std::vector<int> data(initData);          // BAD - Uses copy construction.
      std::vector<int> data(std::move(initData));  //  OK - Uses move construction.
  }

  struct MyMovableType {
      std::string d_data;
      /* ... */
      MyMovableType(MyMovableType&& other) noexcept :
          // d_data(other.d_data)                  // BAD - Uses copy construction.
          d_data(std::move(other.d_data))          //  OK - Uses move construction.
      { /* ... */ }
  };
```
Function templates that have *forwarding-reference* parameters – i.e., those of form `TYPE&&` (where `TYPE` is deduced) and can therefore operate on arguments passed by either lvalue or rvalue reference due to reference collapsing rules – require (instead of `std::move`) the use of `std::forward` to propagate the argument's value category (i.e., to perform *perfect forwarding*):
```
  void func(SpecificType&);       // overload (1) - accepts lvalue reference
  void func(SpecificType&&);      // overload (2) - accepts rvalue reference

  template<class TYPE>
  void func_wrapper(TYPE&& init) {
      // func(init);              // BAD - Always uses the lvalue-ref overload (1).
      // func(std::move(init));   // BAD - Always uses the rvalue-ref overload (2).

      func(std::forward<TYPE>(init));
        // OK - Uses overload (1) for lvalues and overload (2) for rvalues.
  }
```
Finally, the state of a *moved-from object* is typically unspecified; hence, calling a function, such as `std::string::front`, having a *narrow contract* (i.e., one with preconditions) on a moved-from object might lead to *undefined behavior*[10].

noexcept

The `noexcept` specifier enables the explicit denotation of functions as being non-throwing, thereby enabling *qualitative* (semantically significant) algorithmic optimizations – most notably those pertaining to preferring efficient moves over inefficient copies in mutating container operations, such as `std::vector::push_back`, that offer the *strong* exception-safety guarantee. Decorating a non-throwing function with `noexcept` might also improve local code generation for its clients by enabling compilers to elide stack-unwinding code in the calling context without needing visibility into the function's implementation:
```
  int calledFn() noexcept;  // The implementation of 'calledFn' is not visible,
                            // but it is explicitly marked 'noexcept'.
  int callingFn() {
      return calledFn();    // No exceptions can be thrown by 'calledFn'; hence, no
  }                         // code is generated in 'callingFn' to handle them.
```
Such local code changes to functions that never throw are, however, only *quantitative* (i.e., do **not** affect program semantics), and their effect on performance

[10] The essential validity of moved-from objects is usually maintained including *self*-move assignment – e.g., `x = std::move(x);` – see http://ericniebler.com/2017/03/31/post-conditions-on-self-move/. Hence, invoking a function having a *wide contract* on a moved-from object is typically safe.

(if any) is typically negligible. Furthermore, `noexcept` might precipitate additional (exception-detection) code[11] in the function to which it is applied:

```
int nonNoexceptFn();
int calledFn() noexcept {
    return nonNoexceptFn();  // The compiler must add code to call 'std::terminate'
}                            // (just in case 'nonNoexceptFn' actually throws).
```

Applying `noexcept` to a function having preconditions (i.e., a *narrow contract*) precludes a contract-violation handler from throwing an exception in response to Defensive-Programing (DP) assertion failures – a behavior that might be useful (1) in production software, or (2) even just as pragmatic means for validating such defensive checks during unit testing. Note that a new, core-language-based Contracts Facility[12] is anticipated for C++20. Also note that the C++ Standard itself has followed the policy[13] of adding `noexcept` only to functions having no preconditions (i.e., a *wide contract*) ever since the introduction of this keyword.

constexpr

The `constexpr` mechanism simplifies expressing compile-time computations and widens their scope to include user-defined types. Limitations placed on `constexpr` functions (especially prior to C++14), however, often preclude implementation of an optimal algorithm, leading to performance penalties if executed at runtime. Such runtime execution *will* occur unless *all* arguments are *constant expressions*, and *might* happen anyway unless the value that the `constexpr` function returns is used in a context requiring a constant expression:

```
constexpr float power(float base, int exp) {  // example-only 'constexpr' function
    return exp == 0 ? 1 : base * power(base, exp - 1);
}

void use_power(float base) {
    constexpr float value = power(1.4f, 100);
      // Guaranteed to be computed at compile time.
    float value = power(1.4f, 100);
      // Will likely be computed at compile time, but there is NO guarantee.

    constexpr float value = power(base, 100);
      // Will not compile - 'base' is not known at compile time.
    float value = power(base, 100);
      // Will be computed at runtime, but much less efficiently
      // than it would have been by simply calling 'std::pow'.
}
```

Separately, complex `constexpr` computations (e.g., sorting an array of pairs to represent a `constexpr map`) – especially if performed in a header file and, hence, in all translation units that include it – might increase compilation time significantly.

Variadic Templates

Variadic templates (previously approximated by a finite suite of non-variadic ones) enable function and type templates to accept an arbitrarily large number of template arguments. Excessive use of such templates, e.g. involving pseudo-recursive implementations, might, however lead to substantially increased (and difficult-to-analyse) compile times, as well as defeat certain compiler optimizations[14]:

[11] Marking a function that makes a potentially throwing call `noexcept` might also preclude tail-call optimization, in which a simple jump instruction is used to avoid allocating a new stack frame.
[12] See P0542R3: Support for contract-based programming in C++ (https://wg21.link/p0542r3).
[13] See N3279: Conservative use of noexcept in the Library (https://wg21.link/n3279).
[14] See Meeting C++ 2015 keynote by Chandler Carruth (video at ~36:00, slides #44).

```
template <typename T, typename ...Ts>
int hash(hash_state &h, T arg, Ts ...args) {
    add_to_state(h, arg);    // some complex computation to put 'arg' into state 'h'
    return hash(h, args...); // Pseudo-recursive instantiation and function call
}                            // that might be problematic for the optimizer.
```

USER-DEFINED LITERALS

User-defined literals enable integer, floating-point, character, and string literals to produce objects of user-defined type via the invocation of user-defined "suffix" operators. To be effective, however, the suffixes must be short, which, unlike function and class names, cannot be disambiguated with qualification in case of a name clash at the point of use – thus requiring copious (locally scoped) `using` directives instead.

DEFAULT MEMBER INITIALIZERS

Default member initializers can help to avoid duplicate initializations in multiple constructors, and allow non-default initialization of non-inherited members in inherited constructors. Any change to the initializer, however, necessarily requires recompilation of all clients, as opposed to only relinking if the initialization is performed in constructors defined entirely within a separate source file.

"FORWARD" DECLARATIONS OF ENUMERATIONS

An *opaque* enumeration declaration enables the use of that enumeration without granting visibility to its enumerators. Unlike a forward `class` declaration, however, an opaque enumeration declaration produces a complete type, sufficient for substantive use (e.g., via the linker). Hence, a *local* opaque `enum` declaration (whose underlying type's consistency with its complete definition cannot be enforced by the compiler) might lead to an ill-formed program (no diagnostic required):

event_component.h:

```
enum class Event : int { /* slowly growing list of enumerators */ };
int processEvent(Event e);
```

evil_application.cpp:

```
// 'event_component.h' is not included to avoid recompilation when enumerators of
// the 'Event' enumeration are added.
enum class Event : int;  // BAD - Local enum declaration: Program silently becomes
                         // ill-formed if underlying type of 'Event' in
// ...                   // 'event_component.h' is changed.
```

If the underlying type of `Event` changes, but the local declaration is not updated accordingly, the (now ill-formed) program will still compile, link, and run, but its behavior silently becomes undefined. Note that this entirely new incarnation of the classic local-declaration pitfall is far more insidious than it might first appear[15].

[15] The maintainability pitfall associated with opaque enumerators is qualitatively more severe than for other external-linkage types, such as a global `int`, in that the ability to elide the enumerators amounts to an *attractive nuisance* wherein a client – wanting to do so and having access to only a single header containing the unelided definition (i.e., comprising the enumerator name, underlying integral type, and enumerator list) – is seduced into providing an elided copy of the `enum`'s definition (i.e., one omitting just the enumerators) locally. Having the library component provide a second (forwarding) header file containing just the opaque declaration of the enumeration (i.e. enumerator name and underlying integral type only), for example, would be one (generally applicable) way to sidestep this often surprisingly insidious maintenance burden:

event_component_fwd.h:

```
// Clients using this header are not recompiled if just the enumerators change.
enum class Event : int;  // OK - Opaque enumeration declaration for all clients.
int processEvent(Event e);
```

event_component.h:

“UNSAFE” FEATURES

(Misnomer aside) there are at least a couple of features whose practical use is sufficiently specific (rare) that they tend to be disproportionately overused in practice:

- The **`final`** specifier unilaterally restricts clients of a class by either preventing a derived class from overriding individual virtual functions, or (if used on a class) forbidding derivation altogether. Although there exist rare cases where `final` is indispensable[16], its liberal use (especially in library software) might unnecessarily restrict clients and preclude legitimate uses without realizing any actual benefit (e.g., in terms of either improved maintainability or performance).
- **Ref-qualified member functions** can occasionally be used (very effectively) to communicate distinctive class semantics, enable certain optimizations, or improve compile-time detection of client misuse. Such benefits, however, might be outweighed by additional costs associated with increased (1) inherent code complexity, (2) required developer training, and (3) frequency of incorrect use –especially in the context of a very large, mature development organization.

CONCLUSION

The modern features added by C++11 and subsequent standards add clarity and expressiveness to the language. Yet experience has shown that, while some features can be used safely in almost all relevant circumstances, other features have the potential for being misused so as to make software harder to understand, costlier to maintain, and more likely to harbor defects. When a language pitfall is identified, there are frequently multiple viable ways to address it, and choosing the best one requires a deep understanding of the problem in isolation, unencumbered by any one solution, as well as the specific development context in which the solution will be applied. Moreover, sometimes the particular solution choice might be entirely arbitrary, and writing code constantly within a given organization, if done well enough, might be significantly better (e.g., more cost effective) than doing it in two or more different ways, even if each of those ways, were they to have been used exclusively from the start, would have been better. By identifying and concisely elucidating just the specific nature of potential pitfalls – in a manner suitable for consumption by individual software developers, as well as those creating local design rules, coding standards, or recommended best practices – we hope to garner consensus as to “where there be dragons” and thereby help others to realize much of the potential productivity benefits afforded by Modern C++, while avoiding inartful usage that might have a long-term negative impact.

```
#include <event_component_fwd.h>
  // Ensures consistency of enumeration's opaque declaration and (complete) definition.
enum Event : int { /* slowly growing list of enumerators */ };
```

[16] See, e.g., https://akrzemi1.wordpress.com/2012/09/30/why-make-your-classes-final/