0.1 C++11

Small amount of intro text here. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

0.1.1 Attributes

An *attribute* is an annotation (e.g., of a statement or named **entity**) used to provide supplementary information that does not affect the semantics[1] of a well-formed program.

Description

Developers are typically aware of information that is not deducible directly from the source code within a given translation unit. Some of this information might be useful to certain compilers, say, to inform diagnostics or optimizations. Customized annotations targeted at external (e.g., *static analysis*) tools[2] might benefit as well.

C++ attribute syntax C++ supports a standard syntax for attributes, introduced via a matching pair of `[[` and `]]`, the simplest of which is a single attribute represented using a simple identifier, e.g., `attribute_name`:

```
[[attribute_name]]
```

A single annotation can consist of zero or more attributes:

```
[[]]          // Permitted in every position where any attribute is allowed.
[[foo, bar]]  // Equivalent to [[foo]] [[bar]].
```

An attribute may have an (optional) argument list consisting of zero or more syntactically valid (but otherwise arbitrary) comma-separated arguments:

```
[[attribute_name()]]           // Same at attribute_name
[[deprecated("too ugly")]]     // Single-argument attribute
[[theoretical(1, "two", 3.0)]] // Multiple-argument attributes
```

Note that having an incorrect number of arguments or an incompatible argument type is a compile-time error for all standard attributes; the behavior for all other attributes, however, is *implementation-defined* (see *Potential Pitfalls*, below).

Any attribute may be *namespace qualified*[3] (using any arbitrary identifier):

[1]By *semantics* here we typically mean any observable behavior apart from runtime performance. There are, however, cases where an attribute is used such that it will not affect the behavior of a *correct* program, but might affect the behavior of a well-formed yet incorrect one (see *Use Cases*, below).

[2]Such *static analysis* tools include Google’s sanitizers, Coverity, and other proprietary, open-source, and commercial products.

[3]Although attributes having a namespace-qualified name (e.g. `[[gnu::const]]`) are only **conditionally supported**, they have historically been supported on major compilers including both Clang and GCC.

```
[[gnu::const]]  // (GCC-specific) namespace-gnu-qualified const attribute
[[my::own]]     // (user-specified) namespace-my-qualified own attribute
```

C++ attribute placement Attributes can, in principle, be introduced almost anywhere within the C++ syntax to annotate almost anything including an *entity*, *statement*, *code block*, and even entire *translation unit*; however, compilers do not typically support anything resembling arbitrary placement of attributes[4] outside of a *declaration statement*. In some cases, the syntactic entity to which an unrecognized attribute appertains might not be clear from its syntactic placement alone.

In the case of a declaration statement, however, the intended entity is well specified; an attribute placed in front of the statement applies to every entity being declared, whereas an attribute placed immediately after the named entity applies to just that one entity:

```
[[noreturn]] void f(), g();    // Both f() and g() are noreturn.
void u(), v() [[noreturn]]();  // Only v() is noreturn.
```

Attributes placed in front of a declaration statement and immediately behind the name[5] of an individual entity in the same statement are additive (for that entity), as are attributes associated with an entity across multiple declaration statements:

```
[[foo]] void f(), g();  // Declares both f() and g() to be foo.
void f [[bar]](), g();  // Now f() is both foo and bar while
                        //     g() is still just foo.
```

Redundant attributes are not themselves necessarily considered a error; however, redundant standard attributes within the same attribute list might be:

---

[4]An attribute can generally appear syntactically at the beginning of any *statement*, – e.g., `[[attr]] x = 5;` – or in almost any position relative to a *type* or *expression* (e.g., `const int &`) but typically cannot be associated within a named objects outside of a declaration statement:

```
[[]] static [[]] int [[]] a [[]], /*[[]]*/ b [[]];  // declaration statement
```

Notice how we have used the empty attribute syntax `[[]]` above to probe for statically viable positions for arbitrary attributes on the host platform (in this case GCC) – the only invalid one being immediately following the comma, shown above as `/*[[]]*/`. Outside of a declaration statement, however, viable attribute locations are typically far more limited:

```
[[]] void [[]] f [[]] ( [[]] int [[]] n [[]] )
[[]] {
[[]] n /**/ *= /**/ sizeof /**/ ( [[]] const [[]] int [[]] & [[]] ) /**/;
[[]] for ([[]] int [[]] i [[]] = /**/ 0 /**/ ;
/**/ i  /**/ < /**/ n /**/ ;
/**/ ++ /**/ i /**/ )
[[]] {
[[]] ;                        // [[]] denotes viable attribute location
      /**/ }
  /**/ }                                  // /**/ denotes no attribute is allowed
```

Type expressions – e.g., the argument to `sizeof` (above) – are a notable exception.

[5]There are rare edge cases in which an entity (e.g., an anonymous union or `enum`) is “declared”‘ without a name:

```
struct S { union [[attribute_name]] { int a; float b }; };
enum [[attribute_name]] { SUCCESS, FAIL } result;
```

```
[[attr1]] void f [[attr2]](), f [[attr3]](int);
                                          // f()    is attr1 and attr2
                                          // f(int) is attr1 and attr3

[[a1]][[a1]] int x [[a1]][[a1]] &;    // x (the reference itself) is a1

void g [[noreturn]] [[noreturn]]();   // g() is noreturn

void h [[noreturn, noreturn]]();      // Compile-time error: repeated attribute
```

In most other cases, an attribute will typically apply to the statement (including a block statement) that immediately (apart from other attributes) follows it:

```
[[attr1]];                              // null statement
[[attr2]] return 0;                     // return statement
[[attr3]] for (int i = 0; i < 10; ++i); // for statement
[[attr4]] [[attr5]] { /* ... */ }       // block statement
```

The valid positions of any particular attribute, however, will be constrained by whatever entities to which it applies. That is, an attribute such as `noreturn`, that pertains only to functions, would be valid syntactically but not semantically were it placed so as to annotate any other kind entity or syntactic element. Misplacement of standard attributes results in an ill-formed program[6]:

```
void [[noreturn]] g() { throw; }  // Error: appertains to type specifier
void i() [[noreturn]] { throw; }  // Error: appertains to type specifier
```

Common compiler-dependent attributes Prior to C++11, there was no standardized syntax to support conveying such externally sourced information and so non-portable compiler intrinsics (such as `__attribute__((fallthrough))`, which is GCC-specific syntax) had to be used instead. Given the new standard syntax, vendors are now able to express these extensions in a more (syntactically) consistent manner. If an unknown attribute is encountered during compilation, it is ignored, emitting a (likely [7]) non-fatal diagnostic.

The table below provides a brief survey of popular compiler-specific attributes that have migrated to the standard syntax (for additional compiler-specific attributes, see *Further Reading*, below):

The absolute requirement (as of C++17) to ignore unknown attributes helps to ensure portability of useful compiler-specific and external-tool annotations without necessarily having to employ conditional compilation so long as that attribute is permitted at that specific syntactic location by all relevant compilers, but see *Potential Pitfalls*, below.

Use Cases

Eliciting useful compiler diagnostics Decorating entities with certain attributes can give compilers enough additional context to provide more detailed diagnostics. For example,

[6]As of this writing, GCC is lax and merely warns when it sees the standard `noreturn` attribute in an unauthorized syntactic position, whereas Clang (correctly) fails to compile. Hence "creative" use of even a standard attribute might behave differently depending on particular platform.

[7]Prior to C++17, a conforming implementation was permitted to treat an unknown attribute as ill-formed and terminate translation; to our knowledge, however, none of them did.

the `[[gnu::warn_unused_result]]` GCC-specific attribute[8] can be used to inform the compiler (and developers) that a function's return value should not be ignored[9]:

```
struct UDPListener
{
    [[gnu::warn_unused_result]]
    int start();
        // Start the UDP listener's background thread (which can fail for a
        // variety of reasons). Return 0 on success, and a non-zero value
        // otherwise.
};
```

Such annotation of the client-facing declaration can prevent defects caused by a client's forgetting to inspect the result of a function:[10]

```
void init()
{
    UDPListener listener;
    listener.start();           // Might fail - return value must be checked!
    listener.trackPort(27015);  // Possible undefined behavior - BAD IDEA!
}
```

For the code above, `g++ 10.1` produces a useful warning:

```
warning: ignoring return value of 'bool HttpClient::start()' declared
         with attribute 'warn_unused_result' [-Wunused-result]
```

Hinting at better optimizations Some annotations can affect compiler optimizations leading to more efficient or smaller binaries. As an example, decorating the function 'reportError' (below) with the GCC-specific `[[gnu::cold]]` attribute (also available on Clang) tells the compiler that the developer believes the function is unlikely to be called often:

```
[[gnu::cold]] void reportError(const char *message) { /* ... */ }
```

Not only might the definition of `reportError` itself be optimized differently (e.g., for space over speed), any use of this function will likely be given lower priority during branch prediction:

```
void checkBalance(int balance)
{
    if (balance >= 0)  // Likely branch
    {
        // ...
    }
    else  // Unlikely branch
    {
```

[8] For compatibility with `g++`, `clang++` supports `[[gnu::warn_unused_result]]` as well.

[9] The C++17 standard `[[nodiscard]]` attribute serves the same purpose and is portable.

[10] Because the `gnu::warn_unused_result` attribute can in no way affect code generation, it is explicitly *not* ill-formed for a client to make use of an unannotated declaration and yet compile its corresponding definition in the context of an annotated one (or vice versa); such is not always the case, however, and best practice might argue in favor of consistency regardless.

```
        reportError("Negative balance.");
    }
}
```

Because the (annotated) `reportError(const char *)` appears on the else branch of the if statement (above), the compiler knows to expect that `balance` is likely *not* to be negative and therefore optimizes its predictive branching accordingly. Note that even if we are wrong about our guess, the semantics of every well-formed program remain the same.

Delineating explicit assumptions in code to achieve better optimizations Although the presence (or absence) of attributes typically has no effect on the behavior of any well-formed program (beside runtime performance), there are cases where an attribute imparts knowledge to the compiler which, if incorrect, could alter the intended behavior of the program (or perhaps mask defective behavior of an incorrect one). As an example of this more forceful form of attribute, consider the GCC-specific `[[gnu::const]]` attribute (also available on Clang). When applied to a function, this (atypically) powerful (and dangerous, see below) attribute instructs the compiler to *assume* that the function is a *pure function* (i.e., that it always returns the same value for any given set of arguments) and has no *side effects* (i.e., the globally reachable state[11] of the program is unaltered by calling this function):

```
[[gnu::const]] double linearInterpolation(double start, double end, double factor)
{
    return (start * (1.0 - factor)) + (end * factor);
}
```

The `vectorLerp` function (below) performs linear interpolation between two bidimensional vectors. The body of this function comprises two invocations to the `linearInterpolation` function (above) — one per vector component:

```
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

In the (possibly frequent) case where the values of the two components are the same, the compiler is allowed to invoke `linearInterpolation` only once — even if its body is not visible in `vectorLerp`’s translation unit:

```
// Pseudocode (hypothetical compiler transformation)
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    if (start.x == start.y && end.x == end.y)
    {
        const double cache = linearInterpolation(start.x, end.x, factor);
```

[11] Absolutely no external state changes are allowed in a function decorated with `[[gnu::const]]`, including global state changes or mutation via any of the function’s arguments (the arguments themselves are considered local state, and hence can be modified). The (more lenient) `[[gnu::pure]]` allows changes to the state of the function’s arguments, but still forbids any global state mutation. For example, any sort of (even temporary) global memory allocation would be emphatically disallowed.

```
        return Vector2D(cache, cache);
    }

    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

If the implementation of `linearInterpolation` fails to live up to this promise, however, the compiler will not be able to help us and a runtime defect will be the likely result[12].

Using attributes to control external static analysis Since unknown attributes do not prevent a well-formed program from compiling, external static-analysis tools can define their own custom attributes that, while having absolutely no effect on program semantics, can nonetheless be used to embed detailed information to influence or control those tools. As an example, consider the `[[gsl::suppress(/* rules */)]]` Microsoft-specific attribute, which can be used to suppress unwanted warnings from static analysis tools that verify *Guidelines Support Library*[13] rules. In particular, consider GSL C26481 (Bounds rule #1; see **gslrule26481**[14]), which forbids any pointer arithmetic, instead suggesting that users rely on the `gsl::span` type[15]:

```
void hereticalFunction()
{
    int array[] = {0, 1, 2, 3, 4, 5};

    printElements(array, array + 6);  // Elicits warning C26481.
}
```

Any block of code for which validating rule C26481 is considered undesirable can be decorated with the `[[gsl::suppress(bounds.1)]]` attribute:

```
void hereticalFunction()
{
    int array[] = {0, 1, 2, 3, 4, 5};

    [[gsl::suppress(bounds.1)]]          // Suppress GSL C26481.
    {
        printElements(array, array + 6);  // Silence!
    }
}
```

[12]The briefly adopted — and then *unadopted* — contract-checking facility proposed for C++20 contemplated incorporating a feature similar in spirit to `[[gnu::const]]` in which pre-conditions (in addition to being runtime checked or ignored) could be *assumed*; this unique use of attribute-like syntax also required that a conforming implementation could not unilaterally ignore these precondition-checking attributes as that would make attempting to test them result in hard (*language*) *undefined behavior*.

[13]*Guidelines Support Library* is an Open-source library, developed by Microsoft, that implements functions and types suggested for use by the "C++ Core Guidelines" (**cppcoreguidelines**); see **gsl**.

[14]https://docs.microsoft.com/en-us/cpp/code-quality/c26481?view=vs-2019

[15]`gsl::span` is a lightweight reference type that observes a contiguous sequence (or subsequence) of objects of homogeneous type. Useful in interfaces (as an alternative to both pointer/size or iterator pair arguments), and in implementations as an alternative to (raw) pointer arithmetic. Since C++20, the standard `std::span` template can be used instead.

Creating new attributes to express semantic properties Other uses of attributes for static analysis include statements of properties that cannot otherwise be deduced within a single translation unit. Consider a function, `f` that takes two pointers, `p1` and `p2` such that calling the function where `p1` does not refer to an object in the same contiguous block of memory as `p2` is considered a *precondition violation* (as the two addresses are compared internally). Accordingly, we might annotate the function `f` with our own home-grown attribute `in_same_block(p1, p2)`:

```
// lib.h

[[in_same_block(p1, p2)]]
int f(double *p1, double *p2);
```

Now imagine that some client calls this function from some other translation unit but passes in two unrelated pointers:

```
// client.cpp
#include <lib.h>

void client()
{
    double a[10], b[10];
    f(a, b);  // Oops, this is runtime UB
}
```

But, because our static-analysis tool knows from the `in_same_block` attribute that `a` and `b` must point into the same contiguous block, it has enough information to report, at compile time, what might otherwise have resulted in *undefined behavior* at runtime.

Potential Pitfalls

Unrecognized attributes have implementation-defined behavior Although standard attributes work well and are portable across all platforms, the behavior of compiler-specific and user-specified attributes is entirely implementation-defined, with unrecognized attributes typically resulting in compiler warnings.

Such warnings can typically be disabled (e.g., on GCC using `-Wno-attributes`) but then misspellings in even standard attributes will go unreported[16].

Not every syntactic location is viable for an attribute There is a fairly limited subset of syntactic location for which most conforming implementation are likely to tolerate the double-bracketed attribute-list syntax. The ubiquitously available locations include the beginning of any statement, immediately following a named entity in a declaration statement, and (typically) arbitrary positions relative to a *type expression* but, beyond that, caveat emptor.

[16]Ideally there would be a way to silently ignore a specific attribute on a case-by-case bases on every relevant platform.

See Also

See sections `[[noreturn]]` and `[[carries_dependency]]` for a detailed description of the two standard attributes introduced in C++11, and the `[[deprecated]]` section for one introduced in C++14.

Further Reading

https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes

0.1.2 Binary Literals

Integer literals representing their values in base 2.

Description

A *binary literal* (e.g., `0b1010`) – much like a hexadecimal literal (e.g., `0xA`) or an octal literal (e.g., `012`) – is a kind of *integer literal* (in this case, having the *decimal* value `10`). A binary literal consists of a `0b` (or `0B`) prefix followed by a non-empty sequence of binary digits (`0` or `1`): [17]

```
int i        = 0b11110000; // Equivalent to 240, 0360, or 0xF0
const int ci = 0B11110000; // same value as above.
```

The first digit after the `0b` prefix is the most significant one:

```
void f1()
{
    assert( 0 == 0b0);      // 0*2^0
    assert( 1 == 0b1);      // 1*2^0
    assert( 2 == 0b10);     // 1*2^1 + 0*2^0
    assert( 3 == 0b11);     // 1*2^1 + 1*2^0
    assert( 4 == 0b100);    // 1*2^2 + 0*2^1 + 0*2^0
    assert( 5 == 0b101);    // 1*2^2 + 0*2^1 + 1*2^0
    // ...
    assert(42 == 0b11010);  // 1*2^5 + 0*2^4 + 1*2^3 + 1*2^5 + 0*2^4 + 1*2^3
}
```

Leading zeros – as with octal and hexadecimal (but not decimal) literals – are ignored, but can be added for readability:

```
void f2()
{
    assert(  0 == 0b00000000);
    assert(  1 == 0b00000001);
    assert(  2 == 0b00000010);
    assert(  4 == 0b00000100);
    assert(  8 == 0b00001000);
```

[17] Prior to being introduced in C++14, GCC supported binary literals (with the same syntax as the standard feature) as a non-conforming extension since version 4.3; for more details, see **gnu19**, section xyz, pp. 123-456.

```
    assert(256 == 0b10000000);
}
```

The type of a binary literal[18] is by default a (non-negative) `int` unless that value cannot fit in an `int`, in which case its type is the first type in the sequence {`unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`}[19] in which it will fit, or else the program is *ill-formed, diagnostic required*:[20]

```
// Platform 1 - sizeof(int): 4;  sizeof(long): 4;  sizeof(long long) 8;
auto i32  = 0b0111...[ 24 1-bits]...1111;  // i32  is int
auto u32  = 0b1000...[ 24 0-bits]...0000;  // u32  is unsigned int
auto i64  = 0b0111...[ 56 1-bits]...1111;  // i64  is long long
auto u64  = 0b1000...[ 56 0-bits]...0000;  // u64  is unsigned long long
auto i128 = 0b0111...[120 1-bits]...1111;  // i128 is ill-formed/DR
auto u128 = 0b1000...[120 0-bits]...0000;  // u128 is ill-formed/DR

// Platform 2 - sizeof(int): 4;  sizeof(long): 8;  sizeof(long long): 16;
auto i32  = 0b0111...[ 24 1-bits]...1111;  // i32  is int
auto u32  = 0b1000...[ 24 0-bits]...0000;  // u32  is unsigned int
auto i64  = 0b0111...[ 56 1-bits]...1111;  // i64  is long
auto u64  = 0b1000...[ 56 0-bits]...0000;  // u64  is unsigned long
auto i128 = 0b0111...[120 1-bits]...1111;  // i128 is long long
auto u128 = 0b1000...[120 0-bits]...0000;  // u128 is unsigned long long
```

Separately, the precise "starting" type of a binary literal, like any other literal, can be controlled explicitly using the common integer-literal suffixes {`u`, `l`, `ul`, `ll`, `ull`} in either lower- or uppercase:

```
auto i   = 0b101;       // type: int                 value: 5
auto u   = 0b1010U;     // type: unsigned int        value: 10
auto l   = 0b1111L;     // type: long                value: 15
auto ul  = 0b10100UL;   // type: unsigned long       value: 20
auto ll  = 0b11000LL;   // type: long long           value: 25
auto ull = 0b110101ULL; // type: unsigned long long  value: 30
```

Finally, note that affixing a minus sign (`-`) to a binary literal (e.g., `-b1010`) – just like any other integer literal (e.g., `-10`, `-012`, or `-0xa`) is parsed as a non-negative value first, after which a unary minus is applied:

```
void f()
{
    assert(sizeof(int) == 4);            // True on virtually all machines today.
    assert(-10 == -0b1010);              // As if: assert(0 - 10 == 0 - 0b1010);
    assert(0x7fffffff != -0x7fffffff);   // Each literal is an signed int.
    assert(0x80000000 == -0x80000000);   // Each literal is an unsigned int.
}
```

[18]Its *value category* is *prvalue* like every other integer literal.

[19]This same type list applies for both `octal` and `hex` literals but not for decimal literals, which, if initially `signed`, skip over any `unsigned` types, and vice versa (see below).

[20]Purely for convenience of exposition, we make have employed the C++11 `auto` feature to conveniently capture the type implied by the literal itself; for more information, see `auto`.

Use Cases

Bit masking and bitwise operations Prior to the introduction of binary literals, hexadecimal (and before that octal) literals were commonly used to represent bit masks (or specific bit constants) in source code. As an example, consider a function that returns the least significant 4 bits of a given `unsigned int` value:

```
unsigned int lastFourBits(unsigned int value)
{
    return value & 0xFu;
}
```

The correctness of the "bitwise and" operation above might not be immediately obvious to a developer who is not experienced with hexadecimal literals. In contrast, use of a binary literal more directly states our intent to mask all but the four least-significant bits of the input:

```
unsigned int lastFourBits(unsigned int value)
{
    return value & 0b1111u;  // The u literal suffix here is entirely optional.
}
```

Similarly, other bitwise operations such as setting or getting individual bits might benefit from the use of binary literals. For instance, consider a set of flags used to represent the state of an avatar in a game:

```
struct AvatarStateFlags
{
    enum Enum
    {
        e_ON_GROUND     = 0b0001,
        e_INVULNERABLE = 0b0010,
        e_INVISIBLE     = 0b0100,
        e_SWIMMING      = 0b1000,
    };
};

class Avatar
{
    unsigned char d_state;  // Power set of possible state flags

public:
    bool isOnGround() const
    {
        return d_flags & AvatarStateFlags::e_ON_GROUND;
    }

    // ...
};
```

Replicating constant binary data Especially in the context of *embedded development* or emulation, it is not uncommon for a programmer to write code that needs to deal with specific "magic" constants (e.g. provided as part of the specification of a CPU or virtual machine) that must be incorporated in the program's source code. Depending on the original format of such constants, a binary representation can be the most convenient or most easily understandable one.

As an example, consider a function decoding instructions of a virtual machine whose opcodes are specified in binary format:

```
#include <cstdint> // std::uint8_t

void VirtualMachine::decodeInstruction(std::uint8_t instruction)
{
    switch(instruction)
    {
        case 0b00000000u:  // No-op
            break;

        case 0b00000001u:  // add(register0, register1)
            d_register0 += d_register1;
            break;

        case 0b00000010u:  // jmp(register0)
            jumpTo(d_register0);
            break;

        // ...
    }
}
```

Replicating the same binary constant specified as part of the CPU (or virtual machine)'s manual directly in the source avoids the need to mentally convert such constant data to and from, say, a hexadecimal number.

Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

```
int recordCount;
while (cursor.next()) { ++recordCount; }
    //                  ^~~~~~~~~~~~~
    //                Undefined behavior.
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

```
auto recordCount; // Compile-time error.
while (cursor.next()) { ++recordCount; }
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est.

- Donec `laoreet`, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus.
- Proin tempor ac lectus nec elementum. Maecenas augue turpis.
- Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est.

Use Cases

Small amount of intro text can optionally be here. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam.

Avoiding type repetition Sub-section that describes a particular aspect of the feature in concreteterms. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum[21].

```
int x = 10;
auto y = x;
```

Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.[22]

Potential Pitfalls

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.[23]

[21] The relative position of `decltype(range.sort())` in the signature of `sortRangeImpl` is not significant, as long as it is visible to the compiler during template substitution. This particular example (shown in the main text) makes use of a function parameter that is defaulted to `nullptr`. Alternatives involving a trailing return type or a default template argument are also viable:

[22] For more information on foobar, see **lakos20**, section 1.2.3, pp 208-234, especially Figure 1-35, p. 215.

[23] **lakos20**, section 0.5, pp 34-42

```
template <typename Range>
auto sortRangeImpl(Range& range, int) -> decltype(range.sort(), void());
// The comma operator is used to force the return type to void,
      // regardless of the return type of range.sort().

  template <typename Range, typename = decltype(std::declval<Range&>().sort()>
  auto sortRangeImpl(Range& range, int);
      // std::declval is used to generate a reference to Range that can be
      // used in an unevaluated expression
```

Unordered list:

- **Constructor (0)** will be invoked;
- On line **(1)**, execution will be delegated to constructor **(2)**;
- The body of constructor **(2)** will be executed;
- The body of constructor **(1)** will be executed.

Nested unordered list:

- Foo
  - Bar
  - Baz

In the example above, `localhost` will be initialized in the following manner[24]:

1. **Constructor (0)** will be invoked;
2. On line **(1)**, execution will be delegated to constructor **(2)**;
3. The body of constructor **(2)** will be executed;
4. The body of constructor **(1)** will be executed.

Nested ordered list:

1. Foo
   (a) Bar
   (b) Baz

This feature, when used in conjunction with *explicit instantiation definitions*, can significantly improve compilation times for a set of translation units that often instantiate common templates:

Listing 1: code 1

```
void code()
{

}
```

Listing 2: code 2

```
void code()
{

}
```

Attempting to compile `main.cpp` on its own will produce a linker error along the lines of:

[24]Footnote. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Block code in footnote:

```
abcdef
inline
virtual
```

```
undefined reference to Vector2D<float>::normalize()
```

The linker error is expected as the inclusion of `vector2d.h` suppresses implicit instantiation of `Vector2D<float>`. Note that `iVec` is not affected, as the `Vector2D<int>` instantiation does take place.

Readability concerns Using `auto` can hide all information regarding a variable's type, increasing cognitive overhead for the readers. In conjunction with unclear variable naming, disproportionate usage of `auto` can make code unreadable. E.g.

```
int main(int argc, char** argv)
{
    const auto args0 = parseArgs(argc, argv);
        // The behavior of parseArgs is unclear.

    const std::vector<std::string> args1 = parseArgs(argc, argv);
        // It is obvious what parseArgs does.
}
```

While it may be necessary to read `parseArgs`'s contract at least once to fully understand its behavior, an explicit type in the usage site helps readers understand its purpose (see **cwg1655**[25]).

```
testing column width for code
123456789A123456789B123456789C123456789D123456789E123456789F123456789G123456789H
this is 80 characters
```

C++11 introduces three new types of string literal, which provide strong guarantees on the encoding of character sequences:

Encoding	Example	Character Type
UTF-8	`u8"Hello"`	`char`
UTF-16	`u"Hello"`	`char16_t`
UTF-32	`U"Hello"`	`char32_t`

Raw string literals enable developers to embed strings in a program's source code without requiring to escape special character sequences and preserving whitespace, with the goal of enhancing readability. The syntax of the feature is easily understood through an example showing a *regular expression* embedded in source code:

```
//            delimiter and round parenthesis
//                   v~~~             v~~~
const char* regex = R"xxx([0-9]\(".*"\))xxx";
//                  ^     ^~~~~~~~~~~~~
//                  |      string contents
//                  |
//                  | uppercase R
```

[25] http://wg21.link/cwg1655

Lack of interface restrictions In generic code, even if concrete types are dependent on template arguments, `auto` is needlessly lax. It is always possible to identify a *concept*[26] which provides information regarding operations allowed on a type to the reader (see **age86**, pp. 33-35), albeit specifying it in code is cumbersome.[27]

In some particular cases, concepts also carry important semantic meaning that could be lost by using `auto`. E.g.

```
Packet* PacketCache::findFirstCorruptPacket() const
{
    auto it = std::begin(this->d_packet);

    static_assert(IsRandomAccessIterator<decltype(it)>::value,
                  "it must be a random access iterator.");

    return it == std::end(this->d_employees) ? nullptr
                                              : &*it;
}
```

List initialization The meaning of `auto` completely changes when using *list initialization*: `std::initializer_list` is always deduced.

```
auto example0 = 0; // Copy initialiation, deduced as int.
auto example1(0);  // Direct initialiation, deduced as int.
auto example2{0};  // List initialiation, deduced as std::initializer_list<int>.
```

This surprising behavior contradicts the idea of "uniform initialization" and has been widely regarded as a mistake and rectified in C++14.

The `decltype` keyword allows inspecting the declared type of an entity or the type and value category of an expression. What `decltype` yields as the result depends on the provided argument:

- With an unparenthesized *id-expression*[28] or unparenthesized *class member access expression*[29], `decltype` yields the *"declared type"*[30] of the given expression.
- With any other expression of type `T`, `decltype` yields:
  - `T&&` if the value category of the expression is *xvalue*;
  - `T&` if the value category of the expression is *lvalue*;
  - `T` if the value category of the expression is *prvalue*.

Similarly to `sizeof`, the provided expression is not evaluated.

[26] Authors' Note:We will have some footnotes that are authors' notes.
[27] Unless explicitly specified, the *underlying type* of non-strongly typed enumerations is *implementation-defined*.
[28] Footnote. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
[29] Footnote. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
[30] Footnote. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Glossary

word to be defined
Definition text follows. At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga.

word to be defined
Definition text follows. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

word to be defined
Definition text follows.Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus.

word to be defined
Definition text follows. At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga.

Index