C++ value categories and `decltype` demystified

David Mazières

June, 2021

Introduction

Quick C++ quiz:

- Which of the following C++ functions is incorrect and leads to dangerous undefined behavior if you use the return value?

```
decltype(auto)
fn_A(int i)
{
  return i;
}

decltype(auto)
fn_B(int i)
{
  return (i);
}

decltype(auto)
fn_C(int i)
{
  return (i+1);
}

decltype(auto)
fn_D(int i)
{
  return i++;
}

decltype(auto)
fn_E(int i)
{
  return ++i;
}
```

```
decltype(auto)
fn_F(int i)
{
  return (i >= 0 ? i : 0);
}

decltype(auto)
fn_G(int i, int j)
{
  return i >= j ? i : j;
}

struct S {
  int i = 0;
};

decltype(auto)
fn_H()
{
  return (S{});
}

decltype(auto)
fn_I()
{
  return (S{}.i);
}
```

The answer is that `fn_B`, `fn_E`, `fn_G`, and `fn_I` are bad code. The bad functions return references to variables that go out of scope when the function returns. As of this writing, gcc and clang warn about different subsets of the bad functions.

Now obviously all of the above functions, whether buggy or not, should have been written to return `auto` instead of `decltype(auto)`. So is the lesson just to avoid `decltype` because it adds reference types in unintuitive ways? Unfortunately not, because there are other places in the language where you can't avoid the "decltype logic," notably in C++20 requires expressions. Consider the following function:

```
#include <concepts>

// Attempt to add integer to arbitrary type TA
template<typename TA, typename TB> auto
add(TA a, TB b)
  requires requires {
    // ok:
    { a + b } -> std::same_as<TA>;
    // incorrect (should be std::same_as<int&>):
    { b } -> std::same_as<int>;
```

```
  }
{
  return a += b;
}
```

Why is it that `add(1,2)` is ill-formed (though as of this writing accepted by clang)? `add` uses a requires expression to try to limit the types of its arguments. The first occurrence of the `requires` keyword introduces a *requires clause* to restrict what types `TA` and `TB` are acceptable. The second `requires` introduces a *requires expression* inside the requires clause. A requires expression can include several forms of requirement, and here we are using *compound requirements* which state that a particular expression must have a type fulfilling a particular concept. Specifically, we are requiring that `decltype((a+b))` be the same type as `TA` and that `decltype((b))` be the same type as `int`. That's true in the former case but not the latter, because `decltype((b))` is actually the type `int&`.[1]

Examples aside, the real issue here is that C++'s rules for inferring reference types are fairly unintuitive and hard to learn. I attribute the problem to three main factors:

1. Out of reluctance to introduce new keywords, the C++ language committee gave `decltype` two entirely different purposes that are related enough to be confusing and lead to bad typos.
2. Every expression in C++ has both a *type* and a *value category*, and the two are fiendishly non-orthogonal.
3. C++ references are analogous to symbolic links in a file system... except, instead of making symbolic links transparently look like files, C++ makes files transparently look like symbolic links. This isn't a natural way to think of things.

In practice, #1 and #3 are aren't particularly hard to learn, it's just a question of rewiring your intuition to match the twisted logic of the C++ language specification.

#2 is a different story. Every expression has one of three value categories: *lvalue*, *prvalue*, or *xvalue*. The result of `decltype` on expressions is defined to depend on the value category of the expression. Unfortunately, the official definition of value categories is a tangled mess in the language specification. For instance, an informative note in the section introducing value categories optimistically reads:

> The discussion of each built-in operator in [expr.compound] indicates the category of the value it yields and the value categories of the operands it expects. For example, the built-in assignment operators expect that the left operand is an lvalue and that the right operand is a prvalue and yield an lvalue as the result.

From the quote, you might expect to be able to slog through 30 pages of [expr.compound]

[1] Why not just declare `add(TA a, int b)` to take an integer? That might or might not be better. If you want to catch errors by disallowing expressions such as `add("hello", '!')` where the second argument is a type (e.g., `char`) that gets promoted to `int`, then it may be useful to make `b` a template type. (Admittedly, in that case it's still simpler to say `requires std::same_as<TB, int> && requires { { a + b } -> std::same_as<TA>; }`.)

and find an unambiguous specification for the value category of each operand and result of a built-in operator. Or, since life's not perfect, you might at least hope to find a normative requirement that "the built-in assignment operators expect… the right operand is a prvalue." Well if so, you'd be disappointed.[2]

The upshot is that I've spent way too much time staring at rules on the Value categories page of cppreference.com, then trying to substantiate them by clicking through from [expr.prop] to other parts of the standard. I found things organized in a way that was difficult to internalize and remember. And I've hit bugs around value category in both clang and gcc, so I don't just "trust the compiler"—when these bugs are fixed, code will break.

Fortunately, almost every expression has what I'll call an "expression decltype" that unambiguously implies its value category.[3] Moreover, we can reformulate the rules for expression decltype in a way that just subsumes value categories.

There are several reasons expression decltypes are easier to think about than value categories. First, in some cases (such as casts), the expression decltype is explicit and maybe even textually part of the expression. Second, the expression decltypes of built-in operators are analogous to the return types of overloaded operators. Thus, programmers accustomed to operator overloading should find them intuitive (except maybe for the conditional ternary operator `E1 ? E2 : E3`). Finally, while lvalue, prvalue, and xvalue are kind of abstract concepts, you can use the compiler to check the decltype of an expression.

This blog post is my attempt to refactor expression decltype and value categories in a way that is easier to remember and understand. First, I'll go over the official definition of `decltype`. Next, I'll explain and motivate value categories at a high level and explain why they matter. Finally, I'll propose my new and (I hope) easier to internalize algorithm for determining expression decltype.

`decltype` overview

Before getting into value categories, let's review what decltype does. Though there's only one keyword, the syntax `decltype(E)` performs one of two entirely different type calculations depending on `E`:

1. If `E` is an unparenthesized *id-expression* (e.g., `x`, `s.field`, `S::field`), then `decltype(E)` returns the exact type with which the variable, field, or non-type template parameter was declared, including an lvalue or rvalue reference if and only if the variable or field was declared as one. This is a bit like the *lstat(2)* system call, which is one of the few ways to differentiate between files and symbolic links in the file system.

[2] I think the whole [expr.compound] section only makes sense if unspecified operands and results default to prvalue, so in that sense not mentioning the value category of the right-hand operand of `=` could officially make it a prvalue, but I haven't (yet) found any normative language supporting this conjecture.

[3] The exception is the function part of a non-static member function invocation (e.g., `s.method`, `s.*method_ptr`, `p->method`, `p->*method_ptr`). It's not legal to use `decltype` on such expressions—in fact, the only thing you can do is invoke them—but for what it's worth they are considered prvalues.

Let's call this first calculation **variable decltype**, since it gives us the type with which a variable (or field) was declared.

2. If `E` is anything else, including a parenthesized *id-expression* (e.g., `(x)`, `(s.field)`), then C++ makes any reference in `E`'s type completely transparent and undetectable (think *stat(2)*, not lstat). So `decltype(E)` takes the underlying, non-reference type `T` of `E` and decides whether to make it a reference as follows: If `E` is a prvalue, then `decltype(E)` is just `T`; if `E` is an lvalue, then `decltype(E)` is `T&`; and if `E` is an xvalue, then `decltype(E)` is `T&&`.

   Let's call this second calculation **expression decltype**, or, to coin a clunky abbreviation, **exprtype**. Later on, I'll provide an equivalent formulation that does not depend on value categories, in which case we can run the above rule backwards and say an expression `E` is a prvalue if `decltype((E))` is a non-reference type `T`, an lvalue if `decltype((E))` is `T&`, and an xvalue if `decltype((E))` is `T&&`.

The two `decltype` rules are why `fn_A` above (with `return i`) is safe, while `fn_B` (with `return (i)`) is not. The unparenthesized return expression `i` in `fn_A` indicates the use of variable decltype ("lstat"), so the return type is `i`'s exact type, namely `int`. In `fn_B`, by contrast, the return expression `(i)` indicates expression decltype, so C++ says, "I must hide whether or not `i` was declared as a reference, and since I can convert the expression to `int&`, I'll do so regardless of how `i` was actually declared."

There is a rationale for this logic: prvalue expressions such as a literal `int` (`0`) or a function call returning `int` (`getpid()`) cannot be converted to an lvalue reference such as `int&`. Prvalues also cannot be used in certain contexts where an lvalue reference would be valid (e.g., `getpid() = 5` [wrong], or `int *p = &5` [wrong]). By contrast, an lvalue or xvalue can be converted to a prvalue if necessary. In a sense, providing an lvalue reference whenever possible gives you the most powerful applicable type for a given expression. But it's certainly unintuitive until you've learned the rule.

Towards safer use of `decltype`

If the dual-purpose `decltype` keyword seems error-prone and you are concerned about bugs from accidentally using the wrong type calculation, a solution may be to program in a stylized way that protects you from simple errors. There's ample precedent for such an approach. For example, it's now considered bad practice to write "`if (x = y)`"; we write "`if ((x = y))`" to show that, yes, we really intended to do an assignment inside the conditional. When gcc first started warning about assignments in conditionals in the 1990s, I found it offensively paternalistic. In retrospect, the feature has painlessly caught some typos in my code that would have taken a lot more work to debug at runtime.

One easy thing to do would be to define an `exprtype` macro and always use it when you want expression decltype:

```
#define exprtype(E) decltype((E))
```

This at least makes programmer intent explicit: if you aren't calling `exprtype`, you want

variable decltype, not expression decltype. Unfortunately, the dangerous case is generally that you wanted variable decltype and got expression decltype. So what can you do when you don't want expression decltype? Here are a couple of ideas depending on what you really want.

In some cases, you may actually want neither variable nor expression decltype. In particular, you may want the non-reference type of a variable regardless of whether the variable itself was declared as a reference. The type inference for `auto` variables, `auto` return types, and non-reference function-template arguments works this way. The `<type_traits>` header has a `decay` template that emulates the parameter-passing type transformation, essentially removing references and const/volatile qualifications from a type and converting arrays to pointers. To get these rules, you could define:

```
#define autotype(v) std::decay_t<decltype(v)>
```

On the other hand, if you really want variable decltype, you could define a macro specifically for the purpose, like this:

```
#define IGNORE(x)  // causes error if invoked with 2 arguments
#define APPLY_IGNORE(x) IGNORE(x)
#define PARENTHESIZED_TO_COMMA(x) ,
#define vdecltype(v) APPLY_IGNORE(PARENTHESIZED_TO_COMMA v) decltype(v)
```

The `vdecltype` macro causes a compilation error by calling the single-argument `IGNORE` macro with two arguments if the argument `v` is parenthesized.[4] Unfortunately, this won't catch cases such as `vdecltype(++v)`, which would compile and be a reference type, but at least the use of `vdecltype` makes intent clear and would be amenable to static checking.

I don't know the best answer, but it stands to reason that some sort of convention around how to invoke `decltype` for its different purposes could improve the robustness of C++ code.

Value categories

C++ organizes value categories into the hierarchy depicted above [fig:basic.lval]. I'll define and motivate the categories at a high level, but I won't reproduce the categorization rules from the standard because I think it's better to think of value categories as synonymous with reference qualifiers on expression decltypes:

expression decltype	value category
non-reference `T`	prvalue
lvalue reference `T&`	lvalue
rvalue reference `T&&`	xvalue

Nonetheless, understanding value categories at a high level, and understanding the role they play in the language, will help motivate the reference qualifiers on expression decltypes.

[4] See my previous blog post on how macros work if you are wondering why we need `APPLY_IGNORE`.

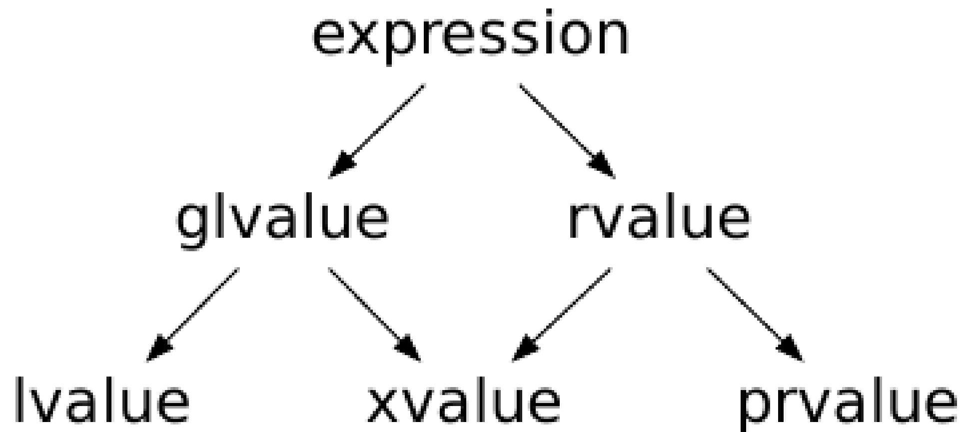

Figure 1: Expression category taxonomy

- As of C++17, a **prvalue** ("pure rvalue") of type `T` is an abstract recipe for initializing an object of type `T` (unless `T` is void). A prvalue does not correspond to an actual object of type `T` in your program. Nor does it require constructor invocation. Literal constants such as `5`, `true`, `nullptr`, and enum tags are prvalues because they initialize objects and operands. For example, the prvalue `5` initializes `x` in "`int x = 5`", initializes the right-hand operand of `+` in "`y + 5`", and initializes the function argument in "`std::to_string(5)`". You cannot modify a prvalue (`true = 1` [wrong]) or take its address (`&this` [wrong]).

  When you write "`auto s = std::string("hello world");`", the cast expression `std::string("hello world")` is a prvalue. Evaluating the prvalue does not create a string object or invoke the `std::string(const char*)` constructor. If it did, `s` would need to be move-constructed from the already-constructed prvalue. Instead, `s` is constructed directly from the argument `"hello world"`. The object that is ultimately initialized by a prvalue (in this case `s`) is known as the prvalue's result object, and the value used to initialize the result object is the prvalue's result.[5]

[5] I find the term "prvalue result" kind of confusing. Obviously the result of prvalue `2+2` is the abstract `int` 4. But what's the result of prvalue `f()` in the following code?

```
struct T {
  std::chrono::high_resolution_clock::time_point point =
    std::chrono::high_resolution_clock::now();
};

T
f()
{
  return T{};
}
```

`f()`'s result is effectively "whatever the current time will be when you eventually materialize this prvalue," which is a strange thing to call a "result."

Compilers try to defer "materializing" prvalues as long as possible to avoid unnecessary moves and copies, particularly when handling function return values. A prvalue must eventually be materialized even if its value is discarded, however, so deferring materialization can elide only copy and move constructors, not other constructors.

- A **glvalue** is an actual object in your program, constructed with a constructor call if its type is not trivially constructible. (The constructor doesn't have to have returned yet.) The specification says a glvalue's "evaluation determines the identity of an object, bit field, or function." This means you can generally take a glvalue's address (except bitfields). You can also assign to a non-const glvalue unless it is a function or a user-defined class with a deleted or inaccessible `operator=`.
- An **xvalue** ("expiring glvalue") is a glvalue whose value will soon not matter, for instance because it is a temporary object about to be destroyed at the end of the current full-expression. Xvalues are what make move construction possible: if you no longer care about the contents of an expiring object, you can often move its contents into another object much more efficiently than if you needed to preserve the expiring object's value. As you might expect, `std::move` transforms its argument into an xvalue.
- An **lvalue** is just a glvalue that is not an xvalue. The archetypal lvalue expression is a variable, but things that behave like variables are lvalues, too, such as class data members and function calls returning lvalue references.
- An **rvalue** is just a prvalue or xvalue.

Why value categories matter

Because a prvalue must be materialized to initialize its result object, it must be a complete type and not an undefined forward-declared struct. Similarly, a prvalue cannot be a pure virtual class, since such classes cannot be constructed other than as superclasses. Also, only class and array prvalues can meaningfully have `const` or `volatile` qualification. There wouldn't be much point in a `const int` prvalue, as it would initialize an `int` lvalue just well as a plain `int` prvalue could. By contrast, a glvalue need not be a complete type and can have `const` and `volatile` qualification for more than just class and array types.

Built-in operators expect particular value categories for their operands and have particular value categories for their results. For example, an arithmetic operator such as binary `+` expects prvalues for both operands and produces a prvalue result. This isn't surprising, since you can supply an integer literal as the argument to `+`, and you also can't assign to or take the address of the result. Built-in `=` expects an lvalue on the left and a prvalue on the right, and results in an lvalue.

Of course, even though `+` has prvalue operands, `x + 1` is still a valid expression when `x` is an lvalue. The reason is that a glvalue other than a function or array can be converted to a prvalue of the same type through a process confusingly named lvalue-to-rvalue conversion. If you recall, a prvalue is really a recipe for initializing an object. When converting from non-class glvalues such as `int`s, this recipe is to initialize the new `int` from the value of the old one. When converting class types, the recipe is to copy- or move-initialize a new instance

of the class from a particular existing glvalue. (This is presumably why class-type prvalues can have `const` and `volatile` qualifiers—they reflect the type of the old glvalue from which the new one should be initialized.)

An expression's value category determines what references the expression may initialize. Specifically, if `T` is a non-reference type, then:

- `T&` can be initialized only from lvalues
- `T&&` can be initialized only from rvalues
- `const T&` can be initialized from any value category, but overload resolution will prefer `T&&` over `const T&` for rvalues if there are functions accepting both. (This is why copy constructors can fill the role of a missing move constructor.)

Note that when binding a prvalue to a reference, it must be materialized into a temporary object. Generally, a temporary object is destroyed at the end of the full expression, which would leave a dangling reference. To avoid this, C++ extends the lifetime of temporary objects that are bound to references, so that they survive until the reference goes out of scope.

The reference binding rules mostly explain which functions can be selected in overload resolution for arguments of what value category. For instance, you can't pass prvalue `5` to a function expecting an `int&`, but you can pass it to a function expecting a `const int&` or an `int&&`. However, there's one rule that doesn't fit the logic, namely a function template argument seemingly expecting a non-const, non-volatile rvalue reference to a simple typename function template parameter:

```
template<typename T> decltype(auto)
f(T&& t)
{
  return g(std::forward<T>(t));
}
```

A function parameter such as `T&& t` is known as a forwarding reference. It matches arguments of any value category, making `t` an lvalue reference if the supplied argument was an lvalue or an rvalue reference if the supplied argument was an rvalue. If `U` is `t`'s underlying non-reference type (namely `std::remove_reference_t<decltype(t)>`), then `T` will be inferred as `U&` for an lvalue argument and `U` for an rvalue. (Through reference collapsing, if `T` is `U&`, then `T&&` is also `U&`.) Regardless of `t`'s variable decltype, its expression decltype is always an lvalue reference; that's why you always need to provide an explicit template argument to `std::forward`.

Note that in the example, `f` actually demonstrates an appropriate use of `decltype(auto)` return type to preserve the value category of `g`'s result (including prvalue). Note also that except for initializer lists, `auto` bindings use the same type deduction rules as function templates. Hence, "`auto &&x = f()`" is another form of forwarding reference.

Now if we treat value categories as synonymous with reference qualification on expression decltypes, then there's a much simpler way to describe the value category of built-in operators.

We can say `+` and `=` behave as if there were built-in functions declared like the following (even though these are obviously not valid code):

```
int operator+(int, int);
int& int::operator=(int);
```

Given that most C++ programmers already understand operator overloading, wouldn't it be clearer to express the value category rules for built-in operators using the same vocabulary as user-defined functions? Expression decltype gets us pretty close to this.

Testing with the compiler

Since value category is just synonymous with the reference qualification on expression decltype, we can get the compiler to tell us the value categories of expressions. All we need to do is invoke expression decltype `decltype((E))`, and use this as a template argument to a template variable that is specialized on reference types. Using this technique, we can figure out which of the functions at the top of this blog post are bad. The function calls that are prvalues are safe, while the lvalues and xvalue are bad, since they are returning references to values that have gone out of scope.

```
template<typename T> constexpr const char *category = "prvalue";
template<typename T> constexpr const char *category<T&> = "lvalue";
template<typename T> constexpr const char *category<T&&> = "xvalue";

#define SHOW(E) std::cout << #E << ": " << category<decltype((E))> << std::endl

int
main()
{
  SHOW(fn_A(0));
  SHOW(fn_B(0));
  SHOW(fn_C(0));
  SHOW(fn_D(0));
  SHOW(fn_E(0));
  SHOW(fn_F(0));
  SHOW(fn_G(0,1));
  SHOW(fn_H());
  SHOW(fn_I());
}
```

output:

```
fn_A(0): prvalue
fn_B(0): lvalue
fn_C(0): prvalue
fn_D(0): prvalue
fn_E(0): lvalue
```

```
fn_F(0): prvalue
fn_G(0,1): lvalue
fn_H(): prvalue
fn_I(): xvalue
```

Here's the code if you want to play with it.

Recall that forwarding references are only for function template arguments, so the specialization `category<T&&>` is not a forwarding reference.

Simplified rules for expression decltype

Here is my recipe for determining `decltype((E))`, the "exprtype" of expression `E`. For the purposes of this blog post, we care less about `decltype((E))`'s underlying non-reference type `T`, and more about whether `decltype((E))` is `T`, `T&`, or `T&&`. Generally `T` itself is either obvious or the result of complicated implicit conversion rules that will have to be the subject of a future blog post.

At a high-level, there are three cases to consider: special-cases, named values, and unnamed values. The special cases are string literals and functions, whose exprtype is always the corresponding lvalue reference `T&`. Named values consist of variables, data members, and array elements. These have exprtype `T&` unless they would be destroyed by destroying some containing struct or array that has non-lvalue-reference exprtype, in which case they have exprtype `T&&`. Finally, unnamed values have the same exprtype as their normal C++ type. Unnamed values can be broken down into two subcases: expressions with an obvious explicit type (e.g., casts and function calls), and ones where you have to think about what a built-in operator does.

I hope the intuition from the previous paragraph is already enough to figure out the vast majority of exprtypes in your head, but since this is all a bit unintuitive and there are some hard cases (ternary operator), let me spell it out in a lot more detail with some added rationale:

- **special lvalues.** Unlike other types, string literals, functions, and references to function always have an exprtype of lvalue reference.

E	decltype((E))
`"hello"`	`const char(&)[6]`
`getpid`	`int(&)()`
`static_cast<int(&)()>(getpid)`	`int(&)()`
`std::move(getpid)`	`int(&)()`

  Because a string literal is a `char[]` in memory and a function is instructions in memory, neither can be a prvalue, so it does not make sense to give them non-reference decltype. Moreover, both string literals and functions have the lifetime of the entire program, so

it does not make sense to move them, which is the point of rvalue references. So that leaves lvalue reference as the only sensible exprtype.

- **named values** If `E` is the value of a variable, a data member in a class or union, or an array member, then it corresponds to a real, constructed object and cannot be a prvalue. Hence, `E`'s exprtype must be a reference. In this case:
  - If `E` resides in an object whose exprtype is not an lvalue reference and `E`'s variable decltype is not a reference, then `E`'s exprtype is an rvalue reference.
  - Otherwise, `E`'s exprtype is an lvalue reference.

  For example:

```
int v;
int &vref = v;
int a[10];

struct S {
  static int static_member;
  int data_member = 0;
  int &lref = static_member;
  int &&rref = std::move(static_member);
};
S s;
S f();
S &lvf();
int S::*fieldp = &S::data_member;
```

E	decltype((E))
v	int&
vref	int&
a	int(&)[10]
a[5]	int&
S::static_member	int&
s.data_member	int&
S::data_member	int&
s.*fieldp	int&
s.lref	int&
s.rref	int&
lvf().data_member	int&
lvf().lref	int&
lvf().rref	int&
f().lref	int&
f().rref	int&
S{}.lref	int&
S{}.rref	int&

E	decltype((E))
S{}.data_member	int&&
S{}.*fieldp	int&&
f().data_member	int&&
f().*fieldp	int&&
std::move(a)[5]	int&&

An unintuitive consequence of this rule is that the reference qualification on `E`'s *expression decltype* is generally independent of that on its *variable decltype*. For example, `v` and `vref` have the same exprtype, as do `s.lref` and `s.rref`. The one exception is inside objects with non-lvalue exprtype. `f().data_member` has exprtype `int&&`, because it resides within `f()`'s return value, which is of non-reference exprtype `S`, so since `f()`'s return value is expiring, so is `f().data_member`. By contrast, `f().lref` and `f().rref` both have `int&` exprtype because the `int`s they reference won't be destroyed when the `S` object returned by `f` is destroyed. To push further on the file system analogy, destroying a class or array is like recursively deleting a directory—only the regular files in the directory will expire, not the files named by symbolic links in that directory.

A tricky case worth explaining is why a non-static data member such as `S::data_member` is always an lvalue. Inside a method of `S`, the expression `S::data_member` is equivalent to `(*this).S::data_member`, which more obviously has exprtype `int&`. Outside of `S`, it is not legal to evaluate `S::data_member`, but the expression is still an lvalue in unevaluated contexts such as `decltype((S::data_member))`. It can even be used inside of prvalue expressions like `decltype(S::data_member + 5)`. (To put this in perspective, it's okay to write `sizeof(S::data_member + 42)` as well.)

- **explicitly typed values**. When an unnamed value expression `E` has an explicit type, `E`'s exprtype is the type of the expression. More specifically:
  - All literals (except string literals), enum tags, `this`, and non-type template arguments have exprtype identical to their type (with no references added).

E	decltype((E))
true	bool
5	int
'A'	char
nullptr	std::nullptr_t
enum tag	corresponding enum type
`this` inside `T::method()`	`T*` or `const T*`

  - Function calls, including overloaded operators, have exprtype identical to the function's return type.

```
int v;
```

E	decltype((E))
std::terminate()	void
std::to_string(5)	std::string
std::move(v)	int&&
co_await a	decltype(a.await_resume())
std::string("hello") + " world"	std::string

- The exprtype of casts and type conversions is exactly the destination type.

E	decltype((E))
double(1)	double
static_cast<int&>(v)	int&
static_cast<int&&>(v)	int&&
std::string{"hello"}	std::string

- `new`'s exprtype is pointer to the type requested.

E	decltype((E))
new T	T*

- **built-in operators**. Implicitly-typed built-in operators have types analogous to how you would overload them for user-defined types.
  - The ternary operator `E1 ? E2 : E3` is the one exception, since you cannot overload it. It attempts to unify the types and `const`/`volatile` qualifiers of the exprtypes of `E2` and `E3`. It will use a common reference type if it can find one, which is why `fn_G` at the beginning of the blog post returns a reference (both `E2` and `E3` have exprtype `int&`, which can be unified). By contrast, `fn_F` unifies `int&` and `int`, which can only be an `int`, not an `int&`, so `fn_F` returns `int`.

    Note: the ternary operator is the only implicitly-typed built-in operator capable of having an rvalue reference exprtype.
  - Pointer dereferencing (`*p`) a pointer of type `T*` has exprtype `T&`.
  - All other built-in operator expressions that don't modify their arguments have a non-reference exprtype identical to the type of the result value. This includes arithmetic (`a+b`, `-a`) and bit (`a|b`, `a<<b`, `~a`) operations, where the exprtype is either identical to the operands or the result of implicit conversion. It includes logical (`a&&b`, `!a`) and comparison (`a==b`) operators, whose exprtype is `bool` (or `std::strong_ordering`/`std::partial_ordering` for `<=>` on non-

floating point/floating point types). It also includes address-of (`&a`) with exprtype `T*` where `T` is the underlying non-reference type of `a`.

- Assignment (`=`, `+=`, etc.) operators have an exprtype of lvalue reference to the type of the left-hand operand's type.
- Pre-increment/pre-decrement (`++c`, `--c`), have an exprtype of lvalue reference to the type of the operand.
- Post-increment and post-decrement (`c++`, `c--`) have exprtype identical to the non-reference type of the operand (since they must return a value that is no longer the value of the operand, this value has no inherent place to live and must be a prvalue).
- `throw` and `delete` have exprtype `void`.
- `sizeof(E)` has exprtype `std::size_t`.
- `typeid(v)` has exprtype `const std::type_info&`.

Conclusion

C++ is notoriously hard to learn in part because it has so many ad-hoc rules that can't be rederived from first principles. Nonetheless, at least some of the complexity arises from poor organization of the language specification, rather than the language itself. This can be fixed if we explain the language differently, particularly when the language itself is improved, as happened with prvalues in C++17.

In this post, I argued that value categories and expression decltype are two concepts that really should be one. I hope that presenting them in a unified way makes them more intuitive and easier to learn by leveraging the understanding most C++ programmers already of operator overloading and function types.