Identifying literal types

Knowing what is and what is not a literal type is not always obvious given all the various rules we have covered and how the rules have changed from one version of the Standard to another. Having a concrete way of identifying literal types other than becoming a language lawyer and interpreting the full Standard definition can be immensely valuable during development, especially when trying to prototype a facility that we intend to be usable at compile time. We identify two means for ensuring that a type is a literal type and, often more importantly for a user, identifying if a type is a **usable literal type**.

1. Only literal types can be used in the interface of a **`constexpr`** function (i.e., either as the return type or as a parameter type), and any literal type can be used in the interface of such a function. The first approach one might take to determine if a given type is a literal type would be to define a function that returns the given type *by value*. This approach has the downside of requiring that the type in question also be *copyable* or at least *movable*; see Section 2.1."*Rvalue* References" on page 710[12]:

```
struct LiteralType    { constexpr LiteralType(int i)    {} };
struct NonLiteralType {           NonLiteralType(int i) {} };
struct NonMovableType { constexpr NonMovableType(int i) {}
                        NonMovableType(NonMovableType&&) = delete; };

constexpr LiteralType    f(int i) { return LiteralType(i);    }  // OK
constexpr NonLiteralType g(int i) { return NonLiteralType(i); }  // Error
constexpr NonMovableType h(int i) { return NonMovableType(i); }  // Error
```

   In the above example, `NonMovableType` is a literal type but is not movable or copyable, so it cannot be the return type of a function. Passing the type as a *by-value* parameter works more reliably and even consistently identifies non*copyable*, non*movable* literal types:

```
constexpr int test(LiteralType t)    { return 0; }  // OK
constexpr int test(NonLiteralType t) { return 0; }  // Error
constexpr int test(NonMovableType t) { return 0; }  // OK
```

   This approach is appealing in that it provides a general way for a programmer to query the compiler whether it considers a given type, `S`, *as a whole* to be a literal type and can be succinctly written[13]:

```
constexpr int test(S) { return 0; }  // compiles only if S is a literal type
```

   ~~Note that all of these tests require providing a function body, since compilers will validate that the declaration of the function is valid for a **`constexpr`** function only~~

[12] As of C++17, the requirement that the type in question be *copyable* or *movable* to return it as a *prvalue* is removed; see Section 2.1."*Rvalue* References" on page 710.

[13] As of C++14, we can **`return void`** — **`constexpr void`** `test(S) { }` — and omit the **`return`** statement entirely; see Section 2.2."**`constexpr`** Functions '14" on page 959.

~~when they are processing the *definition* of the function.~~ A declaration without a body will not produce the expected error for *non-literal-type* parameters and return types:

```
constexpr NonLiteralType quietly(NonLiteralType t);  // OK, declaration only
constexpr NonLiteralType quietly(NonLiteralType t) { return t; }  // Error
```

Finally, the C++11 Standard Library provides a type trait — `std::is_literal_type` — that attempts to serve a similar purpose[14]:

```
#include <type_traits>  // std::is_literal_type
static_assert( std::is_literal_type<LiteralType>::value, "");     // OK
static_assert(!std::is_literal_type<NonLiteralType>::value, "");  // OK
```

The important takeaway is that we can use a trivial test in C++11 (made even more trivial in C++14) to find out if the compiler deems that a given type is a literal type.

2. To ensure that a type under development is meaningful in a compile-time facility, confirming that objects of a given literal type can actually be constructed at compile time becomes imperative. This confirmation requires identifying a particular form of initialization and corresponding **witness arguments** that should allow a user-defined type to assume a valid compile-time value. For this example, we can use the interface test to help prove that our class, e.g., `Lt`, is a literal type:

```
class Lt  // An object of this type can be used in a constant expression.
{
    int d_value;

public:
    constexpr Lt(int i) : d_value(i != 75033 ? throw 0 : i) { }  // OK
};

constexpr int checkLiteral(Lt) { return 0; }  // OK, literal type
```

Proving that `Lt` in the code example above is a usable literal type next involves choosing a **constexpr** constructor (e.g., `Lt(int)`), selecting appropriate witness arguments (e.g., `75033`), and then using the result in a constant expression. The compiler will indicate if our type cannot be constructed at compile time by producing an error:

```
char x[(Lt(75033), 1)];              // OK, usable in constant expr
static_assert((Lt(75033), true), ""); // OK,   "    "     "       "
```

[14]Note that the `std::is_literal_type` trait is deprecated in C++17 and removed in C++20. The rationale is stated in **meredith16**:

> The `is_literal_type` trait offers negligible value to generic code, as what is really needed is the ability to know that a specific construction would produce constant initialization. The core term of a literal type having at least one **constexpr** constructor is too weak to be used meaningfully.