Translation and evaluation

A mental model for compile-time metaprogramming

Document Number: P0992r0
Audience: SG7
Date: Apr 2, 2018
Reply-to: Andrew Sutton <asutton@uakron.edu>

The purpose of this note is to document my understanding of what it means to evaluate an expression at compile time. During my work on reflection and metaprogramming, I've often found myself getting lost in the various interactions between the compiler (or translator) and the evaluation subsystem. In particular, it is not always obvious what information is allowed to flow from the translation to evaluation and back. Value-based reflection and metaprogramming also add new wrinkles to the language, since they move both values and code in between translation and evaluation. Constructing a sound mental model early on is useful. This note should be helpful for anybody writing metaprogramming proposals.

We begin by defining two terms:

- *Translation* is a process concerned with lexical, syntactic, and semantic analysis of source code. The state of a translation maintains data related to the interpretation of text as programs, and includes tokens, ASTs, scopes, lookup tables, etc.
- *Evaluation* (or execution) is a process concerned with computation. The state of an evaluation maintains data related to computation, and contains instruction streams, runtime stores, call stacks, etc.

In the conventional use of C++, these two processes are physically separate: You compile a program, and then later you run the program. You cannot, during program execution, request the instantiation of a template. The translation process is no longer active at the time of execution. (This is not C#.)

The constant expression facility allows a translation to request the evaluation of an expression at particular points in the program. As with the conventional usage scenario, we should not expect to invoke translation facilities (e.g., name lookup, template instantiation, etc.) during evaluation even though the evaluation is happening at compile time. In contrast, template *instantiation* (a commonly used system for metaprogramming) is a subroutine of translation that depends on many of the its other facilities. Don't confuse instantiation with evaluation; while they can both used for metaprogramming, they are fundamentally different.

I've found it to be instructive to consider a simple example and the behavior a hypothetical (conceptual) implementation. Here is a program fragment that requires the evaluation of a constant expression.

```
int a[f(42)];
//----------^
```

The arrow indicates the next token to be analyzed by the translation process. In this case, we have parsed and checked the array declarator and its expression and are about to request the evaluation of the array bound.

Our hypothetical implementation is extremely literal in its interpretation of the model described above. A request to evaluate an expression causes an entirely new program to be generated by the compiler.

This program contains the function definitions and static variables in the original source and a `main` function that evaluates the original expression. For the expression above, the `main` part of that program might look something like this:

```
// ... definitions needed to evaluate f(42) ...

int main(int argc, char* argv[]) {
  try {
    std::cout << f(42) << '\n'; // Evaluate and serialize
  } catch (...) {
    std::cout << "error" << '\n'; // Indicates evaluation failure
  }
}
```

This program is compiled (presumably in a mode that detects all forms of undefined behavior, as required by the rules for `constexpr`) and linked to create a system executable. The evaluation request then creates a new process for the executable, executes it, and waits for it to complete. The output of the program is captured by the translation and deserialized to be interpreted as either the value of the computed expression or the error to be diagnosed.

Again, this is a purely hypothetical compiler. No sane implementation would work this way. The purpose is to establish conceptual bounds on the interactions between translation and evaluation.

It is worth noting that all implementations have some back-channel communication between the translation and evaluation processes. For example, both Clang and GCC provide the evaluator with source code information in order to produce reasonable diagnostics. This is fine; source code information is read-only data that can emitted as part of the program-to-be-executed. Diagnostic codes can then be emitted as output from that program, to be reinterpreted as diagnostics by the compiler.

This model provides a clear picture of the limitations of compile-time evaluation in C++. More importantly, it helps us contextualize questions about features building on top of the `constexpr` facilities. For example, here are some questions that have been discussed in various papes and in committee, and sketches of answers as far as I understand them.

- *How does compile-time logging work?* Logging text could be redirected through `std::cerr` and reformatted by the translator within for use within its diagnostic framework.
- *Where does the data for static reflection come from?* It is compiled into the program as a statically allocated graph to support queries and traversal.
- *What does it mean for a metaprogram to generate code?* Similar to logging; the generated source code can be accepted by the translator and applied at the point the metaprogram was executed.
- *What does it mean to return compile-time* `std::string`? The final sequence of objects in the data structure are serialized as output; the compiler reinterprets that output into a constant-initialized object of static storage (or something similar).

Having a mental model of compile-time evaluation that physically separates it from the process of translation has been extremely helpful for me. In particular, it has helped me understand what is *not* possible (e.g., instantiating a template during evaluation). This helps prune the design space for otherwise large and complex language features. Hopefully, others will find this note helpful as well.