Don't `constexpr` All The Things

Document #:	P2043R0
Date:	2020-01-13
Project:	Programming Language C++ SG7 - Compile-time programming

Reply-to:	David Sankel <dsankel@bloomberg.net>

Contents

1 Abstract

The `constexpr` metaprogramming model, which involves annotating code with indicators of compile-time or run-time suitability, has been steadily increasing in capabilities since its debut in C++11. These efforts have been in support of the stated goal of making metaprogramming accessible to even novice developers. Unfortunately, the simultaneous mixture and stratification of run time and compile time code has resulted in a complicated and unintuitive programming model. Elsewhere Sean Baxter has, disconnected from the committee, designed an extremely compelling alternative that is intuitive, more powerful, and completely implemented. This paper describes the limitations of the `constexpr` metaprogramming model and introduces its persuasive alternative, the circle metaprogramming model.

```
// test.cpp
#include <iostream>

@meta        std::cout << "Hello at compile time!" << std::endl;
int main() { std::cout << "Hello at run time!"     << std::endl; }
```

```
$ circle test.cpp
Hello at compile time!
$ ./test
Hello at run time!
```

2 Introduction

In 2017 SG7, formerly known as the compile-time reflection study group, decided to effectively move C++ away from template metaprogramming and toward `constexpr` metaprogramming. It was a bold move, but it was sorely needed. Use of the template metaprogramming model, which dominated C++ until recent years, was relegated to the use of a few C++ experts that endured learning a "by accident" functional language with libraries like [Boost.MPL]. The hope was that expanding the scope and usage of `constexpr` would allow developers to write metaprogramming code the same way they write runtime code. [P0425R0] summarized the consensus at the time when it stated, "If this can be made to work, this would be the best and most consistent approach to do metaprogramming in C++."

With the direction set, a slew of papers were developed to extend the capabilities of `constexpr` metaprogramming. This included `constexpr` dynamic allocation [P0784R7], virtual functions [P1064R0], try blocks [P1002R1], and nontrivial destructors. The more fundamental additions of `std::is_constant_evaluated()` [P0595R2], `consteval` functions [P1073R3], and consteval variables [P0596R1] came as well.

Each of these papers brought with them, unexpectedly, user-facing tradeoffs that chipped away at the original vision of a metaprogramming model that mimics the run time programming model. These tradeoffs stem primarily from the following `constexpr` metaprogramming design choices:

1. The distinctions between run time, compile time, and run/compile time functions.
2. Detection of the above distinction using an implicit "a posteriori" model.
3. The implicit and subtle interleave between run time and compile time constructs.

Section 3 goes into more detail on this.

Parallel to this standardization effort a single engineer, Sean Baxter, in September 2016 decided to embark upon writing an LLVM-based C++ compiler from scratch to implement a radically different approach to metaprogramming. The resulting compiler, [Circle], is now freely available for evaluation. According to the website *"it's like c++ that went away to train with the league of shadows and came back 15 years later and became batman"*. Circle is C++17 compatible and, aside from its own numerous language additions, includes several extensions that are expected in C++20 (e.g. concepts) and beyond (e.g. pattern matching). Only recently has the C++ standardization committee become aware of Sean's work on Circle.

Sean's approach to metaprogramming is based on a few simple principles:

1. C++ run time code should be executable at compile time without modification.
2. The compiler run time and the application run time should be independent.
3. Promotion of code and data from compile time to run time is accomplished primarily through injection.

The Circle metaprogramming approach has sidestepped much of the complexity that comes with `constexpr` metaprogramming by allowing all code to be executed at compile time and clearly distinguishing between compile-time and run-time execution. The result, in the opinion of the author, is a truly accessible metaprogramming model where other models thus far presented have failed. Circle is described in Sections 4 and 5 and our conclusions are presented in Sections 7 and 8.

3 `constexpr` Metaprogramming and its Drawbacks

In this section we explore the complexity of `constexpr` metaprogramming starting with a basic `constexpr` function and working our way to more complex machinery.

3.1 `constexpr` Functions

Central to `constexpr` metaprogramming is the idea of a `constexpr` function. The intuition is that any `constexpr`-marked function can be evaluated at compile-time as long as its arguments can be evaluated at compile time.

Some C++ functions can be marked as `constexpr` and others cannot.

```
// OK
constexpr int plus(int a, int b) {
  return a + b;
}

// OK
constexpr void fast_zero(std::vector<int> &vec) {
  if(vec.size()) std::memset(vec.data(), 0, vec.size() * sizeof(int));
}

// ERROR (likely, but not guaranteed)
constexpr void fast_zero2(std::vector<int> &vec) {
  std::memset(vec.data(), 0, vec.size() * sizeof(int));
}
```

Here is where the developer-facing complexity begins. While many functions can be marked as `constexpr`, not every function is *appropriately* marked as `constexpr`. The `fast_zero2` function above, for example, is not appropriately marked `constexpr` and the compiler will (optionally) emit an error upon compiling this code. The `fast_zero` function is *also* not appropriately marked `constexpr`, but here the compiler will definitely not emit an error upon compiling this code.

This begs the question, "How does one know when a function can be *appropriately* marked as `constexpr`?". Developers writing `constexpr` metaprograms will surely need to know and the compiler will not always indicate an inappropriate mark.

Here is a first attempt to define appropriately marked `constexpr` function.

> An *appropriately* marked `constexpr` function has the following properties[1]:
>
> — It doesn't include `goto`s.
> — It doesn't include `static` variable declarations.
> — It doesn't include calls to non-`constexpr` functions or inappropriately marked `constexpr` functions.
> — It doesn't include calls to naked `new`
> — It doesn't throw
> — It doesn't include an asm declaration
> — . . . *over a dozen other rules.*

While many of these rules are intuitive for a compiler designer (e.g. disallowing `static` variable declarations), the working programmer will likely need to memorize the list or have a reference handy.

There is one other rule that is important to call out and that is that an appropriately `constexpr` function must have parameter types and a result type that is are `constexpr`-friendly (closely related to "literal" types in the standard). Much like the above rules restrict code that can participate in `constexpr` metaprogramming, the idea of "`constexpr`-friendly types" restricts which types can participate.

[1] See [expr.const] and [dcl.constexpr] in [N4842] for a more complete list.

An `constexpr`-friendly type has the following properties[2]:

— It is one of the simple builtin types, an array of `constexpr`-friendly types, or a reference to a `constexpr`-friendly type, or
— It is an aggregate of `constexpr`-friendly types, or
— It is a class or union with all member functions (including constructors and destructors) appropriately marked `constexpr` and with all its data members `constexpr`-friendly-typed.

At this point it is clear that `constexpr` metaprogramming involves programming in a language that is related to normal C++, but is *distinct* from it in several unobvious ways. Here we come to two of the principle drawbacks of `constexpr` metaprogramming:

1. **`constexpr`-metaprogramming requires learning a new language**. One must learn the rules for the `constexpr` C++ subset before embarking upon metaprogramming. A runtime developer can't just pick it up by learning a few new constructs.
2. **`constexpr` code is viral**. In order to write a `constexpr` function, all of its dependencies must be (deeply) modified to additionally become `constexpr`-friendly. This drastically increases the cost of metaprogramming and looses out on taking advantage of existing runtime software capital.

At this point we've veered far off the path towards "allowing developers to write metaprogramming code the same way they write runtime code". However, things get worse.

3.2 Mixing Compile Time and Run Time Code

Consider the `fast_zero` function we defined above that is inappropriately marked `constexpr`:

```
constexpr void fast_zero(std::vector<int> &vec) {
  if(vec.size()) std::memset(vec.data(), 0, vec.size() * sizeof(int));
}
```

The straightforward fix is to rewrite this without using `std::memset`, which is not marked as `constexpr`.

```
constexpr void fast_zero3(std::vector<int> &vec) {
  for(std::ptrdiff_t i = 0; i < vec.size(); ++i)
    vec[i] = 0;
}
```

`fast_zero3` is appropriately marked `constexpr`, but there's a problem. If this function is used at run time, it could be significantly less efficient. The `constexpr` language cannot express efficient code like run time C++ can.

A seemingly simple solution was proposed in [P0595R2] with the "magic" function `std::is_constant_evaluated()`

```
constexpr void fast_zero4(std::vector<int> &vec) {
  if( std::is_constant_evaluated() )
    for(std::ptrdiff_t i = 0; i < vec.size(); ++i)
      vec[i] = 0;
  else
    std::memset(vec.data(), 0, vec.size() * sizeof(int));
}
```

At first glance this looks somewhat sensible even considering that we're allowing the arbitrary intermingling of two distinct languages. `std::is_constant_evaluated()` simply queries whether we should use `constexpr` language or runtime C++ language when a divergence is needed.

We now need to modify the definition of appropriately marked `constexpr` functions to account for this change:

An *appropriately* marked `constexpr` function has the following properties:

— It doesn't include `goto`s.

[2] See [basic.types] in [N4842] for the whole picture, which is greatly simplified here.

— It doesn't include `static` variable declarations.

It additionally has the following properties when it is evaluated as part of a subexpression of a core constant expression:

— It doesn't ~~include~~ make calls to non-`constexpr` functions or inappropriately marked `constexpr` functions.
— It doesn't ~~include~~ make calls to naked `new`
— It doesn't throw
— It doesn't ~~include~~ execute an asm declaration
— ... *over a dozen other rules.*

The "when it is evaluated as part of a subexpression of a core constant" is every bit as complex as it sounds. Lets say we have some function `g` which is marked `constexpr` and is being invoked from `f`:

```
void f() {
  const int i = g();
}
```

If `g` is appropriately marked `constexpr`, then it will be evaluated at compile time. If it is inappropriately marked `constexpr` (and compiles), then it will be evaluated at run time.

The compiler executes a so-called a posteriori evaluation[3] whereby it attempts to evaluate `g()` assuming `std::is_constant_evaluated()` is set to `true` and if it encounters something illegal, it will instead use a run time execution, with `std::is_constant_evaluated()` set to `false`.

The rules behind a posteriori evaluation even without `std::is_constant_evaluated()` are frequently confounding. Consider the following example:

```
constexpr int f()
  if constexpr (std::is_constant_evaluated()) {
    // slow version
  } else {
    // fast version
  }
}
```

Because the execution point of `std::is_constant_evaluated()` is forced at compilation time, the run time version of `f` will never execute the "fast version" of the code. Furthermore, sometimes even within the same function's evaluation, one `std::is_constant_evaluated()` call can be set to true and another set to false. See this example from [CppConstants]:

```
constexpr int f() {
  const int n = is_constant_evaluated() ? 13 : 17;
  int m = is_constant_evaluated() ? 13: 17;
  char arr[n] = {}; // char [13]
  return m + sizeof(arr);
}
int p = f();     // 26
int q = p + f(); // 56
```

All this complexity needs to be understood in order to generally determine if a function is appropriately marked *constexpr*.

The intermingling between *constexpr* language has gone even further than `std::is_constant_evaluated()` though:

— `unions` with some `constexpr`-friendly fields and other non-`constexpr`-friendly fields are allowed to be used in a core constant expression as long as only the `constexpr`-friendly fields are used.
— A class can be used in a core constant expression as long as none of its non-`constexpr` functions are used.

[3] See David Vandevoorde's excellent C++Now 2019 keynote [CppConstants] for a detailed explanation.

— A base class's non-`constexpr` `virtual` members can be called in a core constant expression as long as they resolve to a `constexpr` derived function.

The extended intermingling between the `constexpr` language and the runtime C++ language has introduced these additional drawbacks:

1. **Understanding core constant expression evaluation is required for `constexpr` metaprogramming, but it is too difficult for engineers.**
2. **One function implemented with two different languages is error prone.** The same function can have different semantics at run time vs. compile time and it is difficult to identify these bugs.
3. **Types with different run time and compile time capabilities is confusing**. The features in the standard encourage this behavior with classes, unions, and type hierarchies.

The complexity at this point is without doubt approaching that of `template` metaprogramming, but the situation gets even worse.

3.3 Grafting on Dynamic Allocation

[P0784R7] adds support for `std::vector` usage in `constexpr` metaprogramming. [CppConstants] provides this example:

```
constexpr vector<int> f() {
  return vector<int>{1, 2, 3};
}

constexpr int g() { return (int)f().size(); }
static_assert(g() == 3);
static_assert(f()[1] == 2);
```

So far, so good. This support was added by introducing a special "compile time" allocator, distinct from `new`/`delete`, that is only in effect when `std::is_constant_evaluated()==true`. Unfortunately, if we want to now initialize a possibly-runtime-used `constexpr` vector using this compile time vector, we get a compilation error:

```
constexpr vector<int> v = f(); // Error, cannot promote vector from compile
                               // time to run time.
```

The solution proposed in [P0784R7], that was ultimately rejected in Evolution, involved a convoluted marking in order to allow data structures to support "promotion" of their compile time variant (which has potentially different storage and semantics) to a run time equivalent.

The addition of dynamic allocation to the `constexpr` language brought these new drawbacks:

1. **Additional, unknown language features are required to make `std::vector` work as expected.** It would be a surprise if they don't significantly increase the complication of C++.
2. **Making an allocating datatype support the `constexpr` language requires making it (and all its dependency types) branch on `std::is_constant_evaluated()`.** This is error prone, complicated, and expensive to develop.

4 Introduction to Circle Metaprogramming

`constexpr` metaprogramming, in the opinion of the authors, has not met its requirements of providing a simple, easy-to-learn, metaprogramming framework for C++. Instead, we've ended up with something that is expert-only and, with further development and promotion, will injure C++'s representation. Fortunately, there is an alternative, Circle.

4.1 The Compile Time C++ Interpreter

The basic premise of Circle is that *all* C++ code can be executed at compile time. Consider the example from the abstract:

```
// test.cpp
#include <iostream>

@meta        std::cout << "Hello at compile time!" << std::endl;
int main() { std::cout << "Hello at run time!"     << std::endl; }
```

```
$ circle test.cpp
Hello at compile time!
$ ./test
Hello at run time!
```

You can right away make the following observations:

— We're outputting text at compile time.
— We're using the standard library at compile time.

Under the hood it looks even better:

— `operator<<` is not annotated with `constexpr` or any other special token.
— `operator<<` does not have a special "compile-time" implementation, nor do any of its dependencies.
— `operator<<`'s implementation is vanilla libstdc++.
— `operator<<`'s core dependencies are being executed from the Linux distribution's libstdc++'s `.so` at compile time.

Interesting, right? Compile time code is not limited functions defined in other translation units. This also works as desired:

```
#include <cmath>
#include <iostream>
double f() { return sin(3) + sin(5); }

@meta std::cout << f() << std::endl; // outputs -0.817804 at compile time
```

Even locally visible functions can invoked at compile time.

4.2 Mixing Compile Time and Run Time

If the compile-time execution cannot impact the generated run time code, Circle's metaprogramming model wouldn't be very interesting. Let's modify the previous example slightly:

```
#include <cmath>
#include <iostream>
double f() { return sin(3) + sin(5); }

int main() {
    constexpr double val = @meta f();
    std::cout << val << std::endl; // outputs -0.817804 at run time
}
```

We only used `constexpr` here to prove that `f()` is getting executed at compile time. `val` could have just as easily been declared `const double` or even plain `double`.

Circle additionally provides your standard control flow constructs at compile-time. The following snippet illustrates loop unrolling.

```
#include <iostream>

int main() {
  @meta for( int i = 0; i < 5; ++i ) {
    std::cout << "Unrolled Loop iteration" << i << std::endl;
  }
}
```

Note that `i` is a compile-time variable. Compile-time variables cannot be modified at runtime:

```
#include <iostream>

int main() {
  @meta for( int i = 0; i < 5; ++i ) {
    ++i; // ERROR
    std::cout << "Unrolled Loop iteration" << i << std::endl;
  }
}
```

If we wanted to modify `i`, we'd need to prefix the modification with an `@meta` to ensure it is done at compile time.

`int`s, `float`s, and the usual suspects have automatic promotion from compile time to run time. This doesn't happen for more complicated things like `std::vector`.

```
#include <algorithm>
#include <vector>

@meta std::vector<int> myvec{ 3, 6, 2, 1 };
@meta std::sort(myvec.begin(), myvec.end());

int main() {
  std::vector<int> v = myvec; // ERROR: only "literal" types can be ported
}
```

This is where Circle's powerful code injection features come into play. Efficient code to initialize a `std::vector` would make use of its list initialization syntax:

```
std::vector<int> v { /*want comma separated list of elements here*/ };
```

Circle has the means to generate a parameter pack from a compile-time `std::vector` object using the `@pack_nontype` keyword. We merely need to expand such a pack within the braces.

```
#include <algorithm>
#include <vector>

@meta std::vector<int> myvec{ 3, 6, 2, 1 };
@meta std::sort(myvec.begin(), myvec.end());

int main() {
    std::vector<int> v {@pack_nontype(myvec)...}; // works as expected
}
```

4.3 Other Injection Tools

Circle has several other code injection tools which allows the compile time execution to impact the generated run time code. A simple one is `@expression` which converts a string into code text.

```
#include <iostream>

int main() {
  int i = @expression("10+12");
  std::cout << i << std::endl; // Outputs 22
}
```

Note that `@expression` can take any `std::string` object which could, for example, be the result of executing a function at compile time.

Code can fortunately be injected at a higher level than strings. A function with return type `@mauto` produces an expression. When it is "called" that expression is expanded in place.

```
#include <iostream>

@meta int j = 0;

@mauto f(int i) {
  @meta ++j;
  return i*i*j;
}

void g()
{
  int x = f(100); // Produces in code: int x = 100*100*0;
  int y = f(100); // Produces in code: int y = 100*100*1;
}
```

Combining these features makes for some interesting possibilities:

```
#include <iostream>

@mauto print(std::string s) {
  return std::cout << @string(s)
                   << "=" << @@expression(s)
                   << std::endl;
}

int main()
{
  int x = 12;
  print("x");   // outputs x=12
  print("x*x"); // outputs x*x=144
}
```

In the above example:

— `@string(s)` is convert s, a `std::string`, into an injected string literal.
— `@@expression(s)` is like `@expression(s)` except it is evaluated in the caller's scope.

4.4 Beyond

We've only scratched the surface of Circle's metaprogramming capabilities and this short introduction doesn't give it justice. circle-lang.org has reference material and extended examples that cover:

1. Compile-time reflection.
2. Demonstration of an implementation of Python's "f-Strings" as a library feature.
3. Usage of 3rd party libraries at compile time.

5 The Circle Metaprogramming Model

The Circle metaprogramming model is based on the idea that any C++ code that can be executed at run time should also be executable at compile time. This basic strategy has several implications that are considered here.

5.1 Compiler Runtime vs. Application Runtime

In order for all run time code to be executable at compile time, the compiler requires its own state for global variables that is distinct from the application's state for global variables. Consider the following code:

```
#include <iostream>

int i = 0;

int f() { return ++i; }

int main()
{
    std::cout << (@meta f()) << std::endl; // Outputs 1
    std::cout << (@meta f()) << std::endl; // Outputs 2
    std::cout << i << std::endl;           // Outputs 0
}
```

The compiler has its own runtime just like the application. For the compiler runtime the global variable `i` gets two modifications, one for each call to `f`. The application runtime, on the other hand, does not see any modifications to `i`.

It is easy to see why this is necessary to meet the goal of allowing for execution of any C++ code at compile time. Execution of a metaprogram has its own execution profile and this should not interfere with the application it is generating.

Does this mean the compiler is required to execute static initialization for every variable in the translation unit? Absolutely not. Consider this example:

```
struct C { C(){ std::cout << "Hello World" << std::endl; } } c;
void f() { c; }
@meta f();

int main() { f(); }
```

This function outputs "Hello World" at compile time, without the `@meta f()` call it outputs nothing. The compiler is only required to statically initialize non-meta objects with static/thread_local storage duration if they are ODR-used by the interpreter. In other words, you only pay for what you use when it comes to compile-time execution.

5.2 Foreign Function Interface

Not all functions that might be called at compile time live in the compiled translation unit or the C++ standard library. In order to execute such functions, Circle allows the compiler invoker to pass a list of compiled `.so` libraries. When the compiler encounters a compile time function invocation that it cannot resolve otherwise, it searches these libraries for the mangled symbol and executes it.

This consequence of Circle's design intent is interesting for the following reasons:

1. Any library with a `.so` can have its functions executed at compile time without modification. This enables a vast amount of software capital to be directly utilized.
2. Compile-time code can be executed with run-time performance. These foreign function calls are executing highly-optimized machine code. This speed greatly expands what is possible for C++ metaprograms.

5.3 Clear Bridge Between Compile Time and Run Time

The refusal to distinguish between code that is capable for run time and that which is capable for compile time forces the developer to clearly point out *when* code should be run in each mode. There are no complex rules around when compile-time execution is happening, if there's a `@meta` in front of it, it's running at compile-time.

This is one of those cases where having the compiler do less work guessing at the developer's intent results in code that is more readable and maintainable for the future.

6 Implications

We've surveyed `constexpr` metaprogramming issues and introduced the Circle metaprogramming model. What are the takeaways?

The first is considering what the world would look like with Circle metaprogramming being standard. This paper was named after the famous "Constexpr All the Things" talk [ConstexprAll]. In that talk, Ben Deane and Jason Turner described their heroic attempt at creating a compile-time JSON parser using `constexpr` metaprogramming and succeeding in doing so. The amount of skill and pure dedication that project required was astronomical. `constexpr` programming has improved since then and doing such a thing these days is open to a wider pool of C++ experts now. With Circle metaprogramming, however, a junior developer can do the same with only the following code which is orders of magnitude faster, shorter, and legible:

```
#include <rapidjson/document.h>

@meta rapidjson::Document d;
@meta d.Parse("{\"project\":\"circle\",\"stars\":10}"
```

The second takeaway is that many features that would otherwise be independently developed fall right out of Circle's model. Consider [P1967R0] which attempts to add language features for embedding strings from files at compile time. Note how idiomatic the following Circle code is that accomplishes this:

```
#include <fstream>
#include <iterator>
#include <string>

@meta std::ifstream f("file.txt");
@meta std::string str(std::istreambuf_iterator<char>(t),
                      std::istreambuf_iterator<char>());

const char file_contents[] = @string(str);
```

Consider also [P0596R1] which attempts to add compile-time output for constexpr metaprogramming using special `constexpr_report` and `constexpr_assert` functions. In the circle model this can be accomplished with the traditional `std::cout` and `assert` assets from the standard library

The fact that the Circle metaprogramming model naturally subsumes or otherwise simplifies all this existing work is a further indication that it is superior to the `constexpr` metaprogramming model.

7 Conclusion

The recent work on extending `constexpr` functionality was a valiant attempt at making metaprogramming accessible to the masses, but it was based on the unfortunate idea that compile time code must be distinguished from run time code. As it is said, a mistake made in the beginning is a mistake indeed.

Fortunately, Sean Baxter has come up with a model that has the potential to deliver on that original promise. Circle metaprogramming, with its basis that all runtime code should be executable at compile time, is simpler, faster, more powerful, and ultimately the way we should go as a committee.

8 Acknowledgements

Thanks first and foremost to Sean Baxter, upon whose work this paper is based. Thanks also to Daveed Vandevoorde, Louis Dionne, and several others for all their effort in not only developing a vision for `constexpr` metaprogramming, but taking the time to explain how it works to me and many others.

9 References

[Boost.MPL] 2002. Boost.MPL.
https://www.boost.org/doc/libs/1_72_0/libs/mpl/doc/index.html

[Circle] 2020. Circle Feature Preview.
https://www.circle-lang.org

[ConstexprAll] Ben Deane and Jason Turner. 2017. Constexpr ALL the things! C++Now.
https://youtu.be/HMB9oXFobJc

[CppConstants] Daveed Vandevoorde. 2019. C++ Constants. C++Now Keynote Talk.
https://www.youtube.com/watch?v=m9tcmTjGeho

[N4842] Richard Smith. 2019. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4842

[P0425R0] Louis Dionne. 2017. Metaprogramming by design, not by accident.
https://wg21.link/p0425r0

[P0595R2] Richard Smith, Andrew Sutton, Daveed Vandevoorde. 2018. std::is_constant_evaluated.
https://wg21.link/p0595r2

[P0596R1] Daveed Vandevoorde. 2019. Side-effects in constant evaluation: Output and consteval variables.
https://wg21.link/p0596r1

[P0784R7] Daveed Vandevoorde, Peter Dimov,Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde. 2019. More constexpr containers.
https://wg21.link/p0784r7

[P1002R1] Louis Dionne. 2018. Try-catch blocks in constexpr functions.
https://wg21.link/p1002r1

[P1064R0] Peter Dimov, Vassil Vassilev. 2018. Allowing Virtual Function Calls in Constant Expressions.
https://wg21.link/p1064r0

[P1073R3] Richard Smith, Andrew Sutton, Daveed Vandevoorde. 2018. Immediate functions.
https://wg21.link/p1073r3

[P1967R0] JeanHeyd Meneide. 2019. #embed - a simple, scannable preprocessor-based resource acquisition method.
https://wg21.link/p1967r0