Constexpr: a Great Good but Wrong Idea

Yauhen Klimiankou

klimenkov@bsuir.by

Evgeny.Klimenkov@gmail.com

Belarusian State University of Informatics and Radioelectronics

.

IV ISP RAS Open

5th December 2019

Evaluation based compiler optimizations:

- Constant Folding (CF) (prior to 1968)
  compile-time evaluation of simple arithmetic expressions consisting of numeric literals

Compiler Optimizations

Evaluation based compiler optimizations:

- Constant Folding (CF) (prior to 1968)
  compile-time evaluation of simple arithmetic expressions consisting of numeric literals
- Compile-Time Function Execution (CTFE) (2007)
  compile-time evaluation of function calls

Evaluation based compiler optimizations:

- Constant Folding (CF) (prior to 1968)
  compile-time evaluation of simple arithmetic expressions consisting of numeric literals
- Compile-Time Function Execution (CTFE) (2007)
  compile-time evaluation of function calls
- Compile-Time Evaluations (CTE)
  any computations involving constant data known at compile-time and which can be performed by compiler without effect on the program behaviour

- **C#**
  No CTFE support, CF is explicitly required, employs many more semantic rules and performs many more compile-time checks

CTFE in Programming Languages

- **C#**

  No CTFE support, CF is explicitly required, employs many more semantic rules and performs many more compile-time checks

- **Haskell**

  CTE based on lifting in TemplateHaskell extension, explicit boundary between CTE and RTE

CTFE in Programming Languages

- **C#**
  No CTFE support, CF is explicitly required, employs many more semantic rules and performs many more compile-time checks
- **Haskell**
  CTE based on lifting in TemplateHaskell extension, explicit boundary between CTE and RTE
- **Scala**
  CTE based on macros receiving AST and producing AST which will replace the original one

CTFE in Programming Languages

- **C#**

  No CTFE support, CF is explicitly required, employs many more semantic rules and performs many more compile-time checks

- **Haskell**

  CTE based on lifting in TemplateHaskell extension, explicit boundary between CTE and RTE

- **Scala**

  CTE based on macros receiving AST and producing AST which will replace the original one

- **Rust**

  CTFE support is similar to the approach taken by C++, only pure and deterministic functions using limited language facilities are CTFE-eligible

CTFE in Programming Languages

- **C#**

  No CTFE support, CF is explicitly required, employs many more semantic rules and performs many more compile-time checks
- **Haskell**

  CTE based on lifting in TemplateHaskell extension, explicit boundary between CTE and RTE
- **Scala**

  CTE based on macros receiving AST and producing AST which will replace the original one
- **Rust**

  CTFE support is similar to the approach taken by C++, only pure and deterministic functions using limited language facilities are CTFE-eligible
- **D**

  Explicit support of CTFE, but does not introduce any language idioms for that purposes

C++: Template-Based Metaprogramming

- 1989: Stroustrup has proposed a concept of templates
  aim – add parametric polymorphism to C++, source aim – enhancement of standard library by containers support (introduction of STL)

C++: Template-Based Metaprogramming

- 1989: Stroustrup has proposed a concept of templates
  aim – add parametric polymorphism to C++, source aim – enhancement of standard library by containers support (introduction of STL)
- 1994: Erwin Unruh has discovered template-based CTFE
  Template-based CTFE was introduced in C++ unintentionally and, in fact, is side effect of C++ means for parametric polymorphism

C++: Template-Based Metaprogramming

- 1989: Stroustrup has proposed a concept of templates
  aim – add parametric polymorphism to C++, source aim – enhancement of standard library by containers support (introduction of STL)
- 1994: Erwin Unruh has discovered template-based CTFE
  Template-based CTFE was introduced in C++ unintentionally and, in fact, is side effect of C++ means for parametric polymorphism
- Templates have unintentionally added a new Turing complete functional programming language for CTFE inside C++.

C++: Template-Based Metaprogramming

- 1989: Stroustrup has proposed a concept of templates
  aim – add parametric polymorphism to C++, source aim – enhancement of standard library by containers support (introduction of STL)
- 1994: Erwin Unruh has discovered template-based CTFE
  Template-based CTFE was introduced in C++ unintentionally and, in fact, is side effect of C++ means for parametric polymorphism
- Templates have unintentionally added a new Turing complete functional programming language for CTFE inside C++.
- C++ community has many hackers. What could be better for a hacker than using means for purposes for which they are not intended? → Template Metaprogramming discipline.

C++: Template-Based Metaprogramming

- 1989: Stroustrup has proposed a concept of templates
  aim – add parametric polymorphism to C++, source aim – enhancement of standard library by containers support (introduction of STL)
- 1994: Erwin Unruh has discovered template-based CTFE
  Template-based CTFE was introduced in C++ unintentionally and, in fact, is side effect of C++ means for parametric polymorphism
- Templates have unintentionally added a new Turing complete functional programming language for CTFE inside C++.
- C++ community has many hackers. What could be better for a hacker than using means for purposes for which they are not intended? → Template Metaprogramming discipline.
- Template Metaprogramming is discouraged in the industrial programming domain. Reasons: code is hard to write, read, understand, debug, and maintain.

C++: Generalized Constant Expression

- 2003: Gabriel Dos Reis has shared his thoughts on the topic.

C++: Generalized Constant Expression

- 2003: Gabriel Dos Reis has shared his thoughts on the topic.
- 2006: Reis and Stroustrup have formulated the concept of Generalized Constant Expressions. The proposal introduces a new specifier – **constexpr**.

C++: Generalized Constant Expression

- 2003: Gabriel Dos Reis has shared his thoughts on the topic.
- 2006: Reis and Stroustrup have formulated the concept of Generalized Constant Expressions. The proposal introduces a new specifier – **constexpr**.
- Motivation is an extension of constant folding mechanisms:
  - Embarrassments with numeric limits constants
  - Convoluted bitmask types
  - Fragile enumerated types

C++: Generalized Constant Expression

- 2003: Gabriel Dos Reis has shared his thoughts on the topic.
- 2006: Reis and Stroustrup have formulated the concept of Generalized Constant Expressions. The proposal introduces a new specifier – **constexpr**.
- Motivation is an extension of constant folding mechanisms:
  - Embarrassments with numeric limits constants
  - Convoluted bitmask types
  - Fragile enumerated types
- However, on practice, **constexpr** is commonly considered as a mean for hand-driven developer-guided code optimization.

C++: Generalized Constant Expression

```
constexpr int f(int a, int b)
{
  return a + b;
}

enum E
{
  E_1_5 = f(1, 5) // 1
};

int main()
{
  int buf[f(1, 5)]; // 2

  switch(buf[0])
  {
    case f(1, 5): /* ... */ break; // 3
  }

  return 0;
}
```

C++: Compile-Time Evaluations

```
void C::m()
{
  uint32_t b = 0x00000001;
  uint32_t m = 0xffffffff;

  m += m;

  for(uint32_t i= b + 1; i != 13; i = b + 1)
  {
    b  = i;
    m += m;
  }

  bits = b;
  mask = m;
}
```

Neither **CF** nor **CTFE** is applicable here!
However, this is a **CTE-eligible code**!

C++: critique of **constexpr**

```
constexpr int Sum1(int a, int b)
{
  return a + b;
}

int Sum2(int a, int b)
{
  return a + b;
}

void f()
{
  constexpr int a = Sum2(1, 2);
  constexpr int b = Sum1(3, 4);
            int c = Sum2(5, 6);
            int d = Sum1(7, 8);

  enum D { DC = Sum2(9, 0) };
  enum E { EC = Sum1(1, 9) };
}
```

C++: D counterexample

```
int Sum(int a, int b)
{
  return a + b;
}

void f()
{
    static int a = Sum(1, 2); // compile-time
           int b = Sum(3, 4); // run-time

    enum D
    {
        DC = Sum(7, 8), // compile-time
        EC = Sum(9, 0), // compile-time
    }
}
```

C++: complexity breeds complexity

- C++ de facto is not one but three programming languages:
  - General object-oriented imperative C++.
  - Functional template-based metaprogramming language.
  - Hyper-reduced C++ for generalized constant expressions (GCEC++).

C++: complexity breeds complexity

- C++ de facto is not one but three programming languages:
  - General object-oriented imperative C++.
  - Functional template-based metaprogramming language.
  - Hyper-reduced C++ for generalized constant expressions (GCEC++).
- There are three reasons for GCEC++:
  - Cross-compiler portability of the source code.
  - Complexity of C++ makes it unfriendly for AST-interpretation.
  - Introduction of specifier `constexpr` makes interpreter limitations explicit to the programmer.

Language	Number of Pages
D	311
Rust	417
C	461
PL/I	564
Ada	1221
C++	**2247**

Specifier `constexpr` is a positive optimization hint for compiler. However, the C++ already has an experience of introduction of optimization hints:

- Positive optimization hints:
  - `register` – suggests the compiler to store the variable in a CPU register.

    removed in C+++17 as valueless
  - `inline` – enforces inline expansion optimization and, as a side effect, changes linker behaviour.

    Primary meaning has been lost. Currently works only as linker behaviour modifier.
- Negative optimization hints:
  - `volatile` – prevents optimization of access to a variable and enforces compiler to keep variable value in memory.

    Continues to play an essential role in asynchronous and multithreaded applications.

MISSED LESSON

Programming language design must not contain features which do not add new semantics.

or

Programming language design must not include positive optimization hints or other means dedicated solely for optimization enforcement.

- avalanche of interest to the CTFE in the C++ community.
- expanding of GCEC++ language.
- `if constexpr` has been added in C++17.
- C++20:
  - further expanding of GCEC++ language.
  - further propogation of `constexpr` into C++ standard library.
  - `consteval` – restricts function usage as CTFE-only.
  - `constinit` – restricts variable usage as CTFE-only.

We can observe further growth of language complexity originated by introduction of `constexpr` specifier.

C++: drawbacks

- Usage of `constexpr` for generalization of constant expressions for code generation is extremely uncommon. Actual usage is almost exclusively for code optimization purposes.
- Compiler can perform generalization of constant expressions for code generation transparently (D as an example).
- The specifier `constexpr` introduces weak contract to the language:
  - CTE-eligibility can be deduced by the compiler automatically.
  - Encourages code duplication.
  - Encourages code offloading into headers (longer compilation time, weaker encapsulation).
  - Breaks cross-compiler code portability (the limits for resources involved in CTE-evaluation can not be formally specified).
  - CTE-eligibility is a property of the code, but not of function semantics.
  - Future growth of C++ language complexity.

C++: drawbacks

Code example from P0595R1, 2018-05-04

```cpp
constexpr double power(double b, int x) {
  if (std::is_constant_evaluated() && x >= 0) {
    // A constant-evaluation context: Use a
    // constexpr-friendly algorithm.
    double r = 1.0, p = b;
    unsigned u = (unsigned)x;
    while (u != 0) {
      if (u & 1) r *= p;
      u /= 2;
      p *= p;
    }
    return r;
  } else {
    // Let the code generator figure it out.
    return std::pow(b, (double)x);
  }
}
```

C++: missing features

- Advanced code semantics checks in CTE-context.
- Negative optimization hint for exclusion of aspect-oriented code from evaluation in CTE-context

C++: last but not least argument

We have designed and developed an independent tool, which can be considered to be an external post-build optimizer, which applies CTE to executable binaries, based on ISA-specification of the target platform.

Experience of designing and usage of this tool shows that:

- Completely automatic application of CTE is applicable for industrial use.
- 100% coverage of program code by CTE enforcement is achievable.
- CTE enforcement can be performed in a way bounded neither by the specific programming language in general nor by specific language constructs or abstractions (reduction of CTE to CTFE or CF).
- CTE enforcement can be not bounded by translation unit borders and not prevented by the unavailability of source code.

Questions?