TOOLS AND LIBRARIES FOR COMPILE-TIME SOFTWARE ENGINEERING

Paul KEIR & Joel FALCOU

January 16, 2024

Joel FALCOU

LABORATOIRE INTERDISCIPLINAIRE DES SCIENCES DU NUMÉRIQUE

Context

C++: Language for performances

- C++ has been designed from the start as **close to the metal**.
- *You Don't Pay For What You Don't Use*.
- *Zero Cost Abstractions*.
- C++ carved a piece of the HPC landscape for itself.

Compile-time: An untapped ressource

- Some elements of programs are fully known at compile time but yet computed at runtime.
- Moving those computations at an earlier stage leads to better performance.
- How can we find out those opportunities?
- How can we express these code fragments in a meaningful way?

Compile-time programming in C++

Generative Programming

- Programming is writing process over data.
- Generative programming is writing process over code and program fragments.
- It is a way **to automate code writing**.
- In C++, it often meant **Meta-programming**.

Meta-programming in C++ 03

- Rely on templates functions and classes.
- Embed type or code fragment into reusable components.
- Chant *Cthulhu R'lyeh wgah'nagl fhtagn* to get it working.
- Wait aeons for compilation to end.
- Nobody speaks about error messages.

Compile-time programming in C++

Template based compile-time computation

```cpp
#include <array>

// Unexpected type definition for computing a value
template<int N> struct factorial
{
  // No control statement in template, so recursion is required
  static const int value = N * factorial<N-1>::value;
};

// Recursion terminal case handling
template<> struct factorial<0> { static const int value = 1; };

// Finally, this a block of 5040 integers
std::array<int, factorial<7>::value> data;
```

A Problem of Perspective

The Fundamental Errors of pre-C++11 TMP

- Focus on types.
- Play around silly syntax.
- Low level abstractions.

The Post-C++11 strategy

- Make regular code fragments usable at compile-time.
- Make core meta-programming idioms 1st class citizen.
- Reduce the frontier between compile-time and runtime.

The advent of `constexpr` programming

C++ `constexpr` Through The Ages

Modern C++ Compile-Time Computations

Wider `template` Landscape

- Template type alias **[C++11]**
- Template variable **[C++17]**
- Inline variable **[C++17]**
- Extended Non-Type Template Parameters **[C++20]**

The `constexpr` Challenger

- `constexpr` functions **[C++11/14]**
- `constexpr` lambda **[C++17]**
- `if constexpr` **[C++17]**
- `constexpr` memory **[C++20]**

`constexpr` Functions

C++11 - Trivial functions support (Demo)

```cpp
// Normal looking function
//
// constexpr means : this is acceptable to call in context
// |                 where a compile-time known element is required
// v
constexpr int factorial(int n)
{
  // No local variables
  // No control statement
  // Still have to use recursion
  return n < 2 ? 1 : n * factorial(n-1);
}

// Template integer parameter are suitable compile-time context
std::array<int, factorial(7)> x;
```

constexpr Functions

C++14 - Regular functions support (Demo)

```cpp
// Normal looking function
//
// constexpr means : this is acceptable to call in context
// |                 where a compile-time known element is required
// v
constexpr int factorial(int n)
{
  // Local variables
  int r = 1;

  // Control statement
  for(int i=1;i<=n;i++) r *= i;
  return r;
}

// Template integer parameter are suitable compile-time context
std::array<int, factorial(7)> x;
```

`constexpr` Functions

C++14 - Errors Handling (Demo)

```cpp
constexpr int factorial(int n)
{
  // Calling a runtime only function in constexpr context stops compilation.
  assert(n >= 0);

  int r = 1;
  for(int i=1;i<=n;i++) r *= i;
  return r;
}

// Valid compilation
std::array<int, factorial(7)> x;

// Compilation error
std::array<int, factorial(-3)> x;
```

constexpr Functions

C++14 - Interaction with templates (Demo)

```cpp
template<typename... Types>
constexpr std::size_t largest_size()
{
  std::size_t sizes[] = { sizeof(Types)... };
  std::size_t size = 0;

  for(std::size_t i = 0; i< sizeof...(Types);++i)
    size = size < sizes[i] ? sizes[i] : size;

  return size;
}

auto sz = largest_size<int,char,char[9],void*,float>();
```

`constexpr` Functions

C++17 - More `constexpr` standard components (Demo)

- Algorithms are now `constexpr`
- All obvious compile-time knowable functions are now `constexpr`
- Glaring missing components: `cmath` functions :(

```
template<typename... Types>
constexpr std::size_t largest_size()
{
  std::size_t sizes[] = { sizeof(Types)... };
  return *std::max_element(&sizes[0], &sizes[0]+sizeof...(Types));
}

auto sz = largest_size<int,char,char[9],void*,float>();
```

constexpr Variables

C++11/14 - Variables as constexpr entity (Demo)

- Variable can be defined as `constexpr`.
- They can be either used as regular variable or in other `constexpr` contexts.

```cpp
template<typename... Types>
constexpr std::size_t largest_size()
{
  std::size_t sizes[] = { sizeof(Types)... };
  return *std::max_element(&sizes[0], &sizes[0]+sizeof...(Types));
}

// sz is still usable as a compile-time entity
constexpr auto sz = largest_size<int,char,char[9],void*,float>();
```

`constexpr` Variables

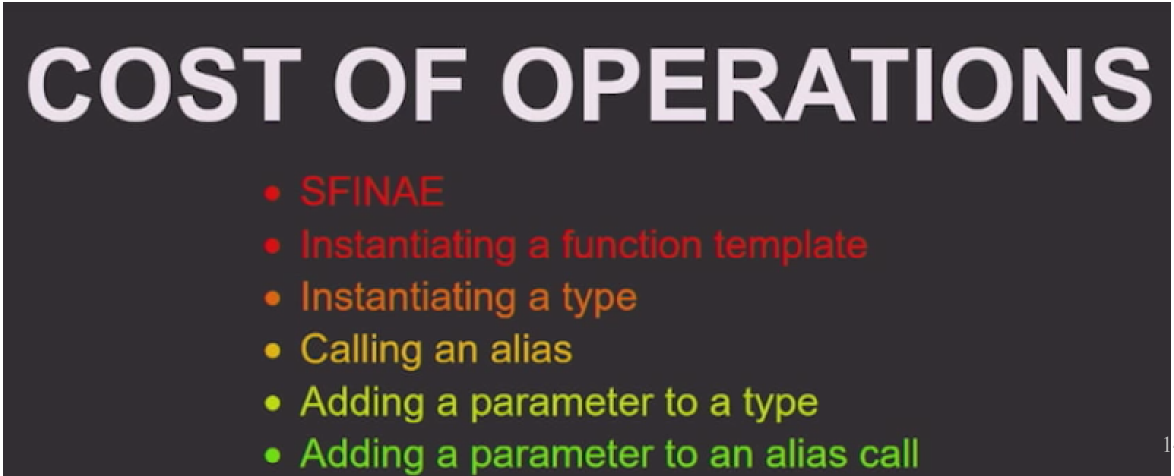
C++17 - Functions as Traits - Take II (Demo)

```
template<typename... Types>
constexpr std::size_t largest_size_impl()
{
  std::size_t sizes[] = { sizeof(Types)... };
  return *std::max_element(&sizes[0], &sizes[0]+sizeof...(Types));
}

template<typename... Types>
struct largest_size
      : std::integral_constant<std::size_t,largest_size_impl<Types...>()>
{};

constexpr auto sz = largest_size<int,char,char[9],void*,float>::value;
```

Template Variables - The Rule of Chiel

Relative costs of template machinery

- C++Now 2017 - Odin Holmes



COST OF OPERATIONS

- SFINAE
- Instantiating a function template
- Instantiating a type
- Calling an alias
- Adding a parameter to a type
- Adding a parameter to an alias call

Template Variables

C++17 - Functions as Traits (Demo)

```cpp
template<typename... Types>
constexpr std::size_t largest_size_impl()
{
  std::size_t sizes[] = { sizeof(Types)... };
  return *std::max_element(&sizes[0], &sizes[0]+sizeof...(Types));
}

// Template variable definition
template<typename... Types>
constexpr auto largest_size_v = largest_size_impl<Types...>();

// Retrieving the value
auto sz = largest_size_v<int,char,char[9],void*,float>;
```

Inline Variables

C++17 - Functions as type_traits

- Solves the multiple definition issue across TU

```cpp
template<typename... Types>
constexpr std::size_t largest_size_impl()
{
  std::size_t sizes[] = { sizeof(Types)... };
  return *std::max_element(&sizes[0], &sizes[0]+sizeof...(Types));
}

// Template variable definition
template<typename... Types>
inline constexpr auto largest_size_v = largest_size_impl<Types...>();

// Retrieving the value
auto sz = largest_size_v<int,char,char[9],void*,float>;
```

Compile-time Code Selection

`std::enable_if` [C++11]

- Substitution failure of template functions leads to removal of functions.
- `std::enable_if` allows us to control this failure

```cpp
template<typename T>
std::enable_if<std::is_trivially_copyable_v<T>> copy(T const* src, T* dst, int n)
{
  std::memcpy(dst,src,sizeof(T)*n);
}

template<typename T>
std::enable_if<!std::is_trivially_copyable_v<T>> copy(T const* src, T* dst, int n)
{
  for(int i = 0;i<n;++i) dst[i] = src[i];
}
```

Compile-time Code Selection

`if constexpr` [C++17]

- `if constexpr` masks branches of code at compile-time.
- Faster to compile.
- Looks like *runtime* code.

```cpp
template<typename T> void copy(T const* src, T* dst, int n)
{
  if constexpr(std::is_trivially_copyable_v<T>)
  {
    std::memcpy(dst,src,sizeof(T)*n);
  }
  else
  {
    for(int i = 0;i<n;++i) dst[i] = src[i];
  }
}
```

Compile-time Memory

`constexpr` Allocations and Containers (Demo)

```cpp
constexpr std::vector<std::string_view> split(std::string_view in, std::string_view d)
{
  std::vector<std::string_view> output;
  std::size_t first = 0;

  while (first < in.size())
  {
    auto second = in.find_first_of(d, first);
    if (first != second)
      output.emplace_back(in.substr(first, second-first));

    if (second == std::string_view::npos) break;
    first = second + 1;
  }

  return output;
}
```

Beyond `constexpr`: `consteval`

`consteval` functions are immediate (Demo)

```cpp
#include <cassert>

struct param
{
  // This constructor must be implicit
  consteval param(int v) :value(v) { assert(v≠0); }
  int value;
};

int f(param a, param b)
{
  return a.value / b.value;
}

auto x = f(8,3);  // Fine
auto y = f(8,0);  // Won't compile
```

Conclusion

`constexpr`: The TMP Savior

Benefits

- Code looks and feels more natural
- Less mental burden when looking unknown code
- Better compile times

What's Next ???

- More lax usage of `constexpr`memory
- New idioms are proposed regularly
- Play with it and Innovate!

Thanks for your attention !