PUBLIC

Lightning Talk: Toying with constexpr and type inference

Xavier Roche
Search Core Team

algolia

Hello!

Xavier Roche

Algolia

Search/Core

algolia

PUBLIC

Fast and safe code with modern C++ ?

Overview

Toying with constexpr & type containers!

1. switch/case for strings
2. runtime function arg validator

PUBLIC

switch/case for strings!

The problem

```
if (key == "poney") {
} else if (key == "elephant") {
} else if (key == "dog") {
} else if (key == "kitten") {
```

- **Linear complexity O(chars)**
- **Not so readable**

The problem

```
if (len == 5 && key == "poney") {
} else if (len == 7 && key == "elephant") {
} else if (len == 3 && key == "dog") {
} else if (len == 6 && key == "kitten") {
```

- **Still linear complexity (but better)**
- **Even less readable and bug prone (did you spot it ?)**

What we want

```
switch(key) {
case "poney":
  break;
case "elephant":
  break;
case "dog":
  break;
case "kitten":
  break;
}
```

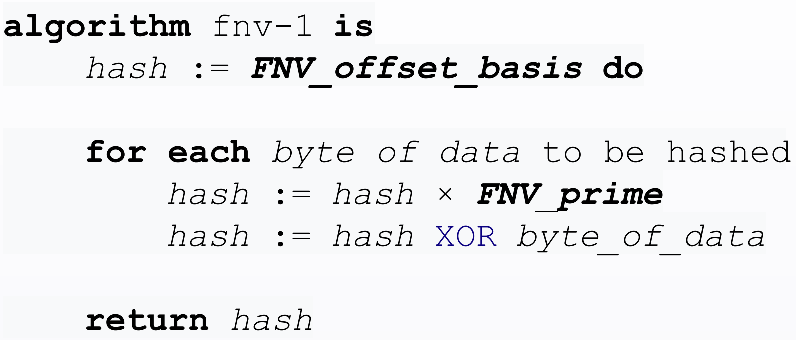
- **Readable**
- **Performances**

What we can do: hash everything!

```
switch(hash(key)) {
case "poney"_hash:
  break;
case "elephant"_hash:
  break;
case "dog"_hash:
  break;
case "kitten"_hash:
  break;
}
```

- **Readable**
- **Performances (hopefully)**

What we need: constexpr fnv-1a/128 hash



```
algorithm fnv-1 is
    hash := FNV_offset_basis do

    for each byte_of_data to be hashed
        hash := hash × FNV_prime
        hash := hash XOR byte_of_data

    return hash
```

WIKIPEDIA
The Free Encyclopedia

- “Good enough” (speed/diffusion)
- Collisions extremely unlikely
- Attacks are possible though

What we need: constexpr hash

PUBLIC

```
template <size_t Bits>
struct fnv1a {
    using Type = typename fnv1a_traits<Bits>::Type;

   /**
    * Compute the Fowler-Noll-Vo hash
    * @param s The string
    * @param l The string size
    * @return The fnv-1a hash
    */
   template <typename C>
   static constexpr Type hash(const C* s, const std::size_t l, Type hash = fnv1a_traits<Bits>::Offset)
   {
       // See <https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function>
       for (std::size_t j = 0; j < l; j++) {
           const uint8_t byte = s[j];
           hash ^= byte;
           hash *= fnv1a_traits<Bits>::Prime;
       }
       return hash;
   }
```

What we need: traits for different hash sizes

```cpp
template <size_t Bits>
struct fnv1a_traits { };

// Traits for 64-bit FNV1a
template <>
struct fnv1a_traits<64> {
    using Type = uint64_t;
    static constexpr Type Prime = 0x100000001b3;
    static constexpr Type Offset = 0xcbf29ce484222325;
};

// Traits for 128-bit FNV1a
template <>
struct fnv1a_traits<128> {
    using Type = __uint128_t;
    static constexpr Type Prime = Pack128(0x1000000, 0x000000000000013b);
    static constexpr Type Offset = Pack128(0x6c62272e07bb0142, 0x62b821756295c58d);
};
```

← 128-bit integer (no std::uint128_t though)

What we need: custom string literal

```
using fnv1a128 = fnv1a<128>;

constexpr auto operator"" _hash(const char* s, const std::size_t l)
{
    return fnv1a128::hash(s, l);
}

// Static unit tests: <https://fnvhash.github.io/fnv-calculator-online/>
static_assert("hello"_hash == Pack128(0xe3e1efd54283d94f, 0x7081314b599d31b3));
```

Cosmetic, but nice!

Hash & dispatch example

algolia

PUBLIC

```
constexpr auto hash(const char* s, const std::size_t l)
{
   return fnv1a128::hash(s, l);
}
```

```
const char* dispatch(const fnv1a_traits<128>::Type hash)
{
    switch (hash) {
    case "poney"_hash:
        return "I want one, too!";
    case "elephant"_hash:
        return "Not in my apartment please!";
    case "dog"_hash:
        return "Good puppy!";
    case "kitten"_hash:
        return "Aawwwwwwww!";
    default:
        return "Don't know this animal!";
    }
  }
```

Dispatch example : hash (extract)

```
label3:
movzbl (%rdi,%r10,1),%r8d
xor    %rax,%r8
mov    %r8,%rax
mul    %r11
shl    $0x18,%r8              ←
add    %rdx,%r8
imul   $0x13b,%rcx,%rbx       ← prime.1
add    %r8,%rbx
movzbl 0x1(%rdi,%r10,1),%ecx
xor    %rax,%rcx
mov    %rcx,%rax
mul    %r11
shl    $0x18,%rcx             ←prime.2
add    %rdx,%rcx
imul   $0x13b,%rbx,%rbx       ←
add    %rcx,%rbx
```

```
   movzbl 0x2(%rdi,%r10,1),%ecx
   xor    %rax,%rcx
   mov    %rcx,%rax
   mul    %r11
→  shl    $0x18,%rcx
   add    %rdx,%rcx
→  imul   $0x13b,%rbx,%rdx
   add    %rcx,%rdx
   movzbl 0x3(%rdi,%r10,1),%ecx
   xor    %rax,%rcx
→  imul   $0x13b,%rdx,%rbx
   mov    %rcx,%rax
   mul    %r11
→  shl    $0x18,%rcx
   add    %rdx,%rcx
   add    %rbx,%rcx
   add    $0x4,%r10
   ...
```

Dispatch example : binary search (extract)

algolia

PUBLIC

```
movabs $0x6efcb17ab10f2a3d,%rax # "kitten"_fnv1a128.1
cmp    %rdi,%rax

movabs $0xfcd05704e13c64bf,%rax # "kitten"_fnv1a128.2
movq   %rdi,%xmm0
movq   %rsi,%xmm1
punpcklqdq %xmm1,%xmm0
sbb    %rsi,%rax
jl     0x400bba <test_kitten>

movdqa 0x17bce(%rip),%xmm1      # "dog"_fnv1a128
pcmpeqb %xmm0,%xmm1
pmovmskb %xmm1,%eax
cmp    $0xffff,%eax
je     0x400bf0 <puppy>
```

```
pcmpeqb 0x17bc7(%rip),%xmm0     # "poney"_fnv1a128
pmovmskb %xmm0,%eax
cmp    $0xffff,%eax
jne    0x400bea <unknown>

mov    $0x41c6a0,%eax           # "I want one, too!"
retq

test_kitten:
movdqa 0x17b7e(%rip),%xmm1      # "kitten"_fnv1a128
pcmpeqb %xmm0,%xmm1
pmovmskb %xmm1,%eax
cmp    $0xffff,%eax
je     0x400bf6 <kitten>

...
```

Want to have a look ?

https://github.com/xroche/stringswitch

algolia

PUBLIC

runtime function arg validator

The problem: size/offset overflow

```
char foo_write(const void*, char size, Foo*);
```

- Function taking non-standard sizes (ie. not std::size_t)
- Real-world is using standard sizes

The problem: size/offset overflow

```
char foo_write(const void*, char size, Foo*);
…
const char str[142] = "Hello!\n";
std::size_t size = 142;
…
const char b = foo_write(str, size, foo);
```

Oops (trivial case here)

What we can do

```
char foo_write(const void*, char size, Foo*);
…
const char str[142] = "Hello!\n";
std::size_t size = 142;
…
assert(size <= std::numeric_limits<char>::max());
const char b = foo_write(str, size, foo);
```

Cumbersome

What we want: automatic cast assert

```
char foo_write(const void*, char size, Foo*);
...
const char b =
    checked_cast_call<foo_write>(str, size, foo);
```

I am lazy:

⟶ **sed -e ‘s/foo_write/checked_cast_call<foo_write>/g’**

Step 1: cast

```cpp
/**
* Template numerical conversion.
* @param v The source numerical value
* @return The destination numerical value, or the original pointer for non-numerical types
* @comment Inspired by Bjarne Stroustrup's in "The C++ Programming Language 4th Edition"
**/
template<class Target, class Source>
auto inline checked_cast(Source v)
{
   if constexpr (std::is_same<Target, Source>::value) {
       return v;
   } else if constexpr (!std::is_integral<Source>::value) {
       return v;
   } else {
       const auto r = static_cast<Target>(v);
       if (static_cast<Source>(r) != v)
           throw std::runtime_error(std::string(__PRETTY_FUNCTION__));
       return r;
   }
}
```

Step 1

```
char foo_write(const void*, char size, Foo*);
...
const char b =
    foo_write(str, checked_cast<char>(size), foo);
```

Need to validate each argument type

Step 2: argument type inference

```
template<typename T>
class checked_cast_call_container
{
public:
    inline constexpr checked_cast_call_container(T result)
        : _result(result)
    {}


    template<typename U>
    inline constexpr operator U() const
    {
        return checked_cast<U>(_result);
    }

private:
    const T _result;
};
```

- T is the “source”
- U is the “target”

← Types inferred

Step 2: argument type inference

```
char foo_write(const void*, char size, Foo*);
...
const char b =
    foo_write(str, checked_cast_call_container(size),
              foo);
```

Need to decorate each argument

Step 3: templated function caller

```
/**
* Wrapped call to a function, with runtime-checked casted input and output values.
* @example checked_cast_call<my_function>(str, 1, size, output)
* @comment This version is using templated function pointer.
*/
template<auto fn, typename... Args>
auto constexpr checked_cast_call(Args... args)
{
   return checked_cast_call_container(fn(checked_cast_call_container(args)...));
}
```

← Non-type template C++17 (`template <typename T, T fn>`)

← Fold expression C++17

We also return a container to infer target

Step 3: templated function caller

PUBLIC

```
char foo_write(const void*, char size, Foo*);
…
const char b =
    checked_cast_call<foo_write>(str, size, foo);
```

Here we are!

Want to have a look ?

algolia

PUBLIC

https://github.com/xroche/checkedcast

https://github.com/xroche/stringswitch
https://github.com/xroche/checkedcast

Questions ?

Show & Tell

Xavier Roche
Search Core Team

algolia