Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 9 – User Defined Literals

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 18.04.2019
FS2019

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR SOFTWARE

Cevelop
Your C++ deserves it

Recap Week 8

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

```cpp
constexpr unsigned pi = 3;
```

- **Evaluated at compile-time (mandatory)**
- **Initialized by a constant expression**
  - Literal value
  - Expression computable by the compiler
  - `constexpr` function calls
- **Require literal type**
- **Can be used in constant expression contexts**
- **Possible contexts**
  - Local scope
  - Namespace scope
  - `static` data members
- **`constexpr` variables are `const`**

- **Instance for `printAll("Hello"s);`**

```
void printAll(std::string const & first) {
  std::cout << first;
  if constexpr (0) {
    std::cout << ", ";
    printAll(); //rest... expansion
  }
}
```

```
void printAll(std::string const & first) {
  std::cout << first;




}
```

- **We dont' need a base case anymore!**

- **Since there is a problem during substitution that overload is discarded**

Name increment is looked up

increment(unsinged)	increment Template

Template arguments are deduced

increment(unsigned)	~~increment<int>(int)~~

Best overload is selected

increment(unsigned)

- **Now the result is 42**

```
unsigned increment(unsigned i) {
  return i++;
}

template<typename T>
auto increment(T value) ->
    decltype(value.increment()) {
  return value.increment();
}

int main() {
  return increment(42);
}
```

- **This approach, using decltype(...) as trailing return type, is infeasible in general**
  - Function might have return type void
- **It is not elegant for complex bodies**

- **You can encode units in value types**
- **You know how to use traits for conversion of units**
- **You can write your own user defined literal operators**
- **(You can compute values at compile-time from raw literals)**

Value Types with Traits

Literal Type for Speed

```
struct Speed {
  constexpr explicit Speed(double kmh)
    : kmh { kmh } {}
  double kmh;
};
```

```
Speed v{1.0}; //Unit?
```

Problem with Literal Values

- **Problem: Literal values lack a dimension**
  - Can result in hard to detect bugs
  - Especially when they are implicitly convertible

```
struct Speed {
  constexpr Speed(double value) {...}
};

bool isFasterThanWalking(Speed speed);
```

```
ASSERT(isFasterThanWalking(10.0));
ASSERT(isFasterThanWalking(2.8));
ASSERT(isFasterThanWalking(6.2));
```

- **Possible Approach: Add factory functions for disambiguation**
  - Tedious to call in many places
  - Difficult to extend (Open Closed Principle is violated)

```
struct Speed {
  static Speed fromKmh(double value);
  static Speed fromMph(double value);
  static Speed fromMps(double value);
private:
  Speed(double value);
};

bool isFasterThanWalking(Speed speed);
```

```
ASSERT(isFasterThanWalking(Speed::fromKmh(10.0)));
ASSERT(isFasterThanWalking(Speed::fromMph(2.8)));
ASSERT(isFasterThanWalking(Speed::fromMps(6.2)));
```

Solution with Tags & Traits (1/3)

- **Create a tag type for the unit**

```
struct Kph;
struct Mph;
struct Mps;
```

- **Create a quantity type template for speed**

```
template <typename Unit>
struct Speed {
  constexpr explicit Speed(double value)
    : value{value}{};
  constexpr explicit operator double() const {
    return value;
  }
private:
  double value;
};
```

- **Add a `speedCast` function**

```
template<typename Target, typename Source>
constexpr Speed<Target> speedCast(Speed<Source> const & source) {
  return Speed<Target>{ ConversionTraits<Target, Source>::convert(source) };
}
```

- **Create a `ConversionTraits` class template**

```
template<typename Target, typename Source>
struct ConversionTraits {
  constexpr static Speed<Target> convert(Speed<Source> sourceValue) = delete;
};
```

- **Specialize `ConversionTraits` class template**

```
template<typename Same>
struct ConversionTraits<Same, Same> {
  constexpr static Speed<Same> convert(Speed<Same> sourceValue) {
    return sourceValue;
  }
};

static constexpr double mphToKphFactor { 1.609344 };
template<>
struct ConversionTraits<tags::Kph, tags::Mph> {
  constexpr static Speed<tags::Kph> convert(Speed<tags::Mph> sourceValue) {
    return Speed<tags::Kph>{static_cast<double>(sourceValue) * mphToKphFactor};
  }
};
//...
```

```cpp
template <typename Unit>
bool isFasterThanWalking(Speed<Unit> speed) {
  return velocity::speedCast<Kph>(speed) > Speed<Kph>{5.0};
}
```

- **Requires comparison operations, i.e >**
  - They can be implemented using boost
- **Arbitrary Speed objects can be compared with an == operator template**

```cpp
template <typename LeftTag, typename RightTag>
constexpr bool operator==(Speed<LeftTag> const & lhs,
                          Speed<RightTag> const & rhs) {
  return lhs == speedCast<LeftTag>(rhs);
}
```

User-Defined Literals

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

- **Type for velocity**

```
template <typename Unit>
struct Speed {
  constexpr explicit Speed(double value)
    : value{value}{};
  constexpr explicit operator double() const {
    return value;
  }
private:
  double value;
};
```

- **Can be used in**

Quite verbose

Example	Valid
`Speed<Unit::kmh> s{5.0};`	Yes
`Speed<Unit::kmh> s = 5.0;`	Non-explicit
`auto s = Speed<Unit::kmh>{5.0}`	Yes
`auto s = 5.0;`	Not a speed object

- **Repetitive occurrence of explicit conversion `Speed<Unit::XYZ>{x}`**

```
auto speed1 = Speed<Kph>{5.0};
auto speed2 = Speed<Mph>{5.0};
auto speed3 = Speed<Mps>{5.0};
```

- **What if we had the possibility to attach units to our literals?**
  - User-defined literals

```
auto speed1 = 5.0_kph;
auto speed2 = 5.0_mph;
auto speed3 = 5.0_mps;
```

- **Overloading**
  - `UDLSuffix` could lexically be any identifier, but must start with underscore _
    (other suffixes are reserved for the standard)
  - Allows to add dimension, conversion, etc.
  - If possible define UDL operator functions as `constexpr`

```
operator"" _UDLSuffix()
```

- **Add the suffix to integer, float and string literals**
  - Suffix belongs to literal (no whitespace between)

```
5.0_kph;  //correct
5.0 _mph; //wrong
```

- **Rule: put overloaded UDL operators that belong together in a separate namespace**
  - Only a `using namespace` can import them

```
using namespace velocity::literals;
```

```
namespace velocity::literals{

constexpr inline Speed<Kph> operator"" _kph(unsigned long long value) {
  return Speed<Kph>{safeToDouble(value)};
}

constexpr inline Speed<Kph> operator"" _kph(long double value) {
  return Speed<Kph>{safeToDouble(value)};
}

//...

}
```

```
void overtakePedestrianAt10Kph() {
  ASSERT(isFasterThanWalking(10.0_kph));
}

void testConversionFromKphToMph() {
  ASSERT_EQUAL(1.60934_kph, 1.0_mph);
}
```

- **Shorter and more expressive literals**
- **`ASSERT_EQUAL` for double already has a margin for its comparison**

- **For literal numbers the following signatures are useful**
  - Integral constants

```
TYPE operator "" _suffix(unsigned long long)
```

  - Example

```
constexpr inline Speed<Kph> operator"" _kph(unsigned long long value) {
  return Speed<Kph>{safeToDouble(value)};
}
constexpr auto speed = 5_kmh;
```

  - Floating point constants

```
TYPE operator "" _suffix(long double)
```

- **For string literals the following signature is useful**

```
TYPE operator "" _suffix(char const *, std::size_t len)
```

```
namespace mystring {
inline std::string operator"" _s(char const *s, std::size_t len) {
  return std::string { s, len };
}
}
```

  - Note: Implementation above cannot be `constexpr`. Why?
  - Example:

```
using namespace mystring;
auto s = "hello"_s;
s += " world\n";
std::cout << s;
```

- **“RAW” UDL Operator**

```
TYPE operator "" _suffix(char const *)
```

```
namespace mystring {
inline std::string operator"" _s(char const *s) {
  return std::string { s };
}
}
```

  - Note: Works only for integral and floating literals, not for string literals!
  - Example:

```
using namespace mystring;
auto rs = 42_s;
rs += " raw\n";
std::cout << rs;
```

Raw UDL Operator Applied

- **Ternary suffix**
  - Base 3
- **Examples**

Ternary	Decimal
0	0
1	1
2	2
10	3
11	4
12	5
20	6
21	7
22	8
100	9

- **Problem: exception at run-time**

```
namespace ternary {
unsigned long long operator"" _3(char const *s) {
  size_t convertedupto{};
  auto res = std::stoull(s, &convertedupto, 3u);
  if (convertedupto != strlen(s))
    throw std::logic_error { "invalid ternary" };
  return res;
}
}
```

```
using namespace ternary;
int four = 11_3;
std::cout << "four is " << four << '\n';
try {
  four = 14_3; // throws
} catch (std::exception const &e) {
  std::cout << e.what() << '\n';
}
```

- **Template UDL Operator**

```
template<char...>
TYPE operator "" _suffix()
```

- **Empty parameter list**
- **Variadic template parameter**
- **Characters of the literal are template arguments**

```
120_ternary; // => operator "" _ternary()
             // with template arguments '1', '2' and '0'
```

- **Unfortunately, the template UDL operator does not work with string literals**

Raw UDL Extreme: Ternary Number at Compile-Time (1/2)

- **Run-time errors for number conversion is bad**
- **There exists a variadic template version of UDL operators**
- **Interpretation of the characters (at compile-time) often requires a variadic class/variable template with specializations**

```
template<char ...Digits>
constexpr unsigned long long operator"" _ternary() {
  return ternary_value<Digits...>;
}
```

- **We will also need a helper function to get the value of the digit: 3^n**

```
constexpr unsigned long long three_to(std::size_t power) {
  return power ? 3ull * three_to(power - 1) : 1ull;
}
```

Raw UDL Extreme: Ternary Number at Compile-Time (2/2)

```
template<char ...Digits>
extern unsigned long long ternary_value;

template<char ...Digits>
constexpr unsigned long long ternary_value<'0', Digits...> {
  ternary_value<Digits...>
};

template<char ...Digits>
constexpr unsigned long long ternary_value<'1', Digits...> {
  1 * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};

template<char ...Digits>
constexpr unsigned long long ternary_value<'2', Digits...> {
  2 * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};

template<>
constexpr unsigned long long ternary_value<>{0};
```

- **Example: `120_ternary`**
- **`120_ternary` -> resolves to `ternary_value<'1', '2', '0'>`**

Partial specialization: `ternary_value<'1', Digits...>`

Value: 1 * 3^2 + `ternary_value<'2', '0'>`

> Partial specialization: `ternary_value<'2', Digits...>`
>
> Value: 2 * 3^1 + `ternary_value<'0'>`
>
> > Partial specialization: `ternary_value<'0', Digits...>`
> >
> > Value: `ternary_value<>`
> >
> > > Partial specialization: `parse_ternary<>`
> > >
> > > Value: 0
> >
> > Value: 0
>
> Value: 6 from 2 * 3^1 + 0

Value: 15 from 1 * 3^2 + 6

- **Can we avoid the duplication of the specialization for `'0'`, `'1'` and `'2'`?**

```
constexpr bool is_ternary_digit(char c) {
  return c == '0' || c == '1' || c == '2';
}

constexpr unsigned value_of(char c) {
  return c - '0';
}

template<char D, char ...Digits>
constexpr
std::enable_if_t<is_ternary_digit(D), unsigned long long>
ternary_value<D, Digits...> {
  value_of(D) * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};
```

- **The declaration of value is barely readable; let's try `static_assert`**
  - `static_assert(cond, msg);`
  - It is a declaration itself and thus cannot be used with variable templates

```
template<char D>
constexpr unsigned value_of() {
  static_assert(is_ternary_digit(D), "Digits of ternary must be 0, 1 or 2");
  return D - '0';
}

template<char D, char ...Digits>
constexpr unsigned long long ternary_value<D, Digits...> {
  value_of<D>() * three_to(sizeof ...(Digits)) + ternary_value<Digits...>
};
```

- **Nice error message during compilation**
- **`static_assert` prevents SFINAE**

- **Upcomming alternative: Concepts**
  - concept keyword
  - Concept name used instead of `typename`

```
template<char D>
concept bool TernaryDigit = is_ternary_digit(D);

template<TernaryDigit D, TernaryDigit...Digits>
constexpr unsigned long long ternary_value<D, Digits...> {...};
```

- **Nice compiler messages**

```
..\main.cpp: In function 'int main(int, char**)':
..\main.cpp:40:27: error: cannot call function 'long long unsigned int ternary::operator""_t()
                                                           [with char ...Digits = {'1', '4'}]'
  std::cout << "14_t: " << 14_t << std::endl;
                           ^~~~
..\main.cpp:30:20: note:   constraints not satisfied
 unsigned long long operator "" _t() {
                    ^~~~~~~~
..\main.cpp:30:20: note:   in the expansion of 'TernaryDigit<Digits>...'
..\main.cpp:30:20: note:     'TernaryDigit<'4'>' was not satisfied
```

Standard Literal Suffixes

- **Standard suffixes don't have a leading underscore**
- **Suffix for `std::string`: `s`**
- **Suffix for `std::complex` (imaginary): `i, il, if`**
- **Suffixes for `std::chrono::duration`: `ns, us, ms, s, min, h`**
- **More might be defined in the future**
- **Is the following example a problem?**

```cpp
using namespace std::string_literals;
using namespace std::chrono_literals;
auto one_s = 1s;
auto one_point_zero_s = 1.0s;
auto fourty_two_s = "42"s;
```

Summary

- **Create your own types if you want to represent values with dimension**
- **User defined literals help giving meaning to simple values**
- **They can be used to compute numbers at compile-time**

Self-Study

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

- **How can we reverse a tuple at compile-time?**
- **Let's try it step by step (TDD approach)**
- **How would you implement `reverse`?**

```
constexpr auto nullary = std::make_tuple();
constexpr auto reversedNullary = reverse(nullary);
static_assert(nullary == reversedNullary);
```

- **Simplest thing that could possibly work:**

```
constexpr auto reverse(std::tuple<> const & nullary) {
  return nullary;
}
```

- **Let's add a second case: Unary tuple**

```
constexpr auto unary = std::make_tuple(1);
constexpr auto reversedUnary = reverse(unary);
static_assert(unary == reversedUnary);
```

- **Add an overload for a tuple with a single template argument**

```
template <typename T>
constexpr auto reverse(std::tuple<T> const & unary) {
  return unary;
}
```

- **Let's add a third case: Binary tuple**

```
constexpr auto binary = std::make_tuple("Hello", 5);
constexpr auto binaryReversed = reverse(binary);
static_assert(binaryReversed == std::make_tuple(5, "Hello"));
```

- **Add an overload for a tuple with a two template arguments**

```
template <typename First, typename Second>
constexpr auto reverse(std::tuple<First, Second> const & binary) {
  return std::make_tuple(std::get<1>(binary), std::get<0>(binary));
}
```

- **Can this scale?**

- **Let's add a fourth case: Ternary tuple**

```
constexpr auto ternary = std::make_tuple("Hello", 5, 3.14);
constexpr auto ternaryReversed = reverse(ternary);
static_assert(ternaryReversed == std::make_tuple(3.14, 5, "Hello"));
```

- **Add an overload for a tuple with a two template arguments**

```
template <typename...Types>
constexpr auto reverse(std::tuple<Types...> const & ternary) {
  return std::make_tuple(std::get<2>(ternary),
                         std::get<1>(ternary),
                         std::get<0>(ternary));
}
```

- **Let's add a fourth case: Ternary tuple**

```
constexpr auto quaternary = std::make_tuple("Hello", 5, 3.14, '*');
constexpr auto quaternaryReversed = reverse(quaternary);
static_assert(quaternaryReversed == std::make_tuple('*', 3.14, 5, "Hello"));
```

- **Add an overload for a tuple with a two template arguments**

```
template <typename...Types>
constexpr auto reverse(std::tuple<Types...> const & nAry) {
  return std::make_tuple(std::get<N - 1>(ternary),
                         std::get<N - 2>(ternary),
                         ...
                         std::get<1>(ternary),
                         std::get<0>(ternary));
}
```

- **Pattern:** `N-1, N-2, N-3, ..., 1, 0`

- **Using non-type template parameters it is possible to encode (integral) values as types**
  - `std::integral_constant<typename INT, INT val>`
  - `std::true_type, std::false_type`

`<type_traits>`

- **Or even whole sequences of integral values**
  - `std::integer_sequence<typename INT, INT...vals>`
  - `std::index_sequence<size_t...vals>`

`<utility>`

- **These classes do not contain data members, they are empty!**
  - but the value can be accessed:
    - `::value`
    - Implicit cast to `INT` (template parameter type)
    - `operator()`

- **Universal reverse:**

```
template <typename Tuple, size_t...Indices>
constexpr auto reverseImpl(Tuple const & nAry, std::index_sequence<Indices...>) {
  constexpr auto firstIndex = sizeof...(Indices) - 1;
  return std::make_tuple(std::get<firstIndex - Indices>(nAry)...);
}

template <typename...Types>
  constexpr auto reverse(std::tuple<Types...> const & nAry) {
  return reverseImpl(nAry, std::make_index_sequence<sizeof...(Types)>());
}
```