Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 7 – Compile-Time Computation

Thomas Corbat / Felix Morgner
Rapperswil, 05.04.2022
FS2022

Agenda

- **Recap Week 6**
- **Compile-time Evaluation with constexpr**
- **Literal Types**
- **User-Defined Literal Operators**

- **Participants should…**
  - … be able to write C++ code that is evaluated by the compiler
  - … know the restrictions of constexpr functions
  - … be able to write their own literal types
  - … be able to write their own user defined literal operators

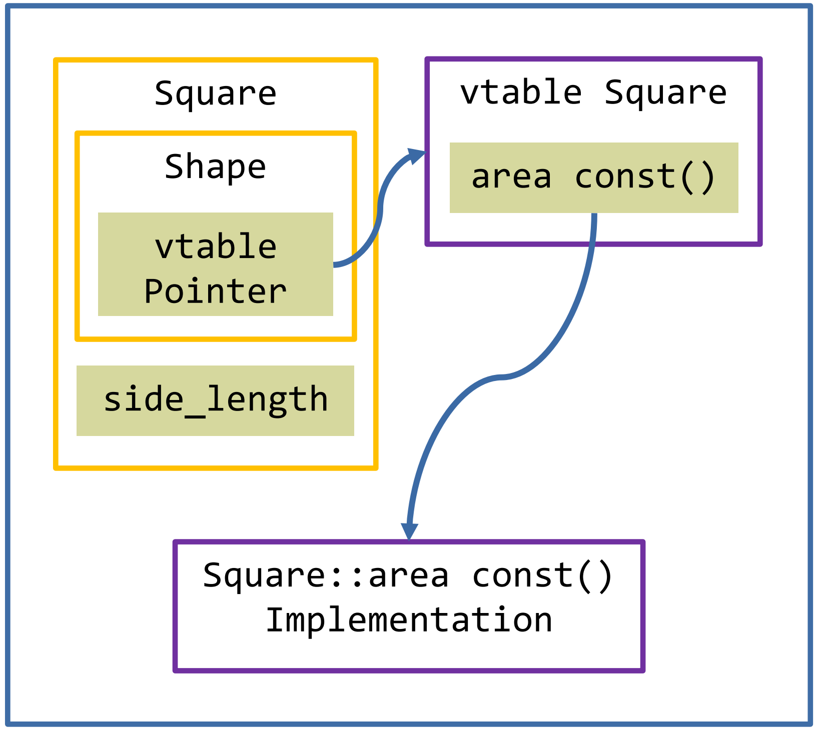
Recap Week 6

- **A polymorphic call of a virtual function requires lookup of the target function**

```
struct Shape {
  virtual unsigned area() const = 0;
  virtual ~Shape();
};

struct Square : Shape {
  Square(unsigned side_length)
    : side_length{side_length} {}
  unsigned area() const {
    return side_length * side_length;
  }
  unsigned side_length;
};
```

```
decltype(auto) amountOfSeeds(Shape const & shape) {
  auto area = shape.area();
  return area * seedsPerSquareMeter;
};
```



User Provided Deduction Guide for `BoundedBuffer<T>`

Template declaration for `Iter`

```
template <typename Iter>
BoundedBuffer(Iter begin, Iter end) -> BoundedBuffer<typename std::iterator_traits<Iter>::value_type>;
```

Constructor signature

Deduced template instance

- **Test for deducing template argument from iterator works**

```
void testDeductionFromIterators() {
  std::vector values{3, 1, 4, 1, 5, 9, 2, 6};
  BoundedBuffer buffer{begin(values), end(values)};
  ASSERT_EQUAL(values.size(), buffer.size());
}
```

- **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

Constexpr

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Basic Features (Legacy)

- **(static) const variables in namespace scope of built-in types initialized with constant expression are usually put into ROMable memory, if at all.**
- **Allowed in constant expression context**
- **No complicated computations (except with macros)**
- **No guarantee to be done at compile-time in all cases**

```
size_t const SZ = 6 * 7;
double x[SZ];
```

Constant expression context

Compile-time evaluation

Constant Expression Contexts

- **Non-type template arguments**

```
std::array<Element, 5> arr{};
```

- **Array bounds**

```
double matrix[ROWS][COLS]{};
```

- **Case expressions**

```
switch (value) {
case 42:
  //...
}
```

- **Enumerator initializers**

```
enum Light {
  Off = 0, On = 1
};
```

- **`static_assert`**

```
static_assert(order == 66);
```

- **`constexpr` variables**

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

- **`constexpr if` statements**

```
if constexpr (size > 0) {
}
```

- **`noexcept`**

```
Blob(Blob &&) noexcept(true);
```

- **...**

```
static_assert(isGreaterThanZero(Capacity));

static_assert(sizeof(int) == 4, "unexpected size of int");
```

- **`static_assert` checked at compile-time**
  - Compilation fails if it evaluates to `false`
  - The compiler needs to be able to evaluate the expression
- **Syntax:**
  - `static_assert(condition);`
  - `static_assert(condition, message);`

```
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`**

```
constexpr auto factorial(unsigned n) {
  ...
}
```

- **`constexpr` functions can…**
  - … have local variables of “literal” type. The variables must be initialized before used.

```
int local;
```

```
LiteralType local{};
```

```
int local;
f(local);
```

```
std::string local{};
```

  - … use loops, recursion, arrays, references
  - … even contain branches that rely on run-time features, if branch is not executed during compile-time computations, e.g., `throw`
  - … but can only call `constexpr` functions

`constexpr` Functions: Restrictions

- **`constexpr` evaluation cannot (yet)...**
  - ... allocate dynamic memory (`new`, `delete`)

```
constexpr int * allocate() {
  return new int{};
}
```

  - ... use exception handling (`throw`, `try`/`catch`)

```
constexpr void throwError() {
  throw std::logic_error{""};
}
```

  - ... be virtual member functions

```
struct Base {
  constexpr virtual void modify();
};
```

- **Constexpr functions are usable in constexpr and non-constexpr contexts**

```
constexpr auto factorial(unsigned n) {
  auto result = 1u;
  for (auto i = 2u; i <= n; i++) {
    result *= i;
  }
  return result;
}

constexpr auto factorialOf5 = factorial(5);

int main() {
  static_assert(factorialOf5 == 120);
  std::cout << factorial(5);
}
```

- **The compiler will prevent Undefined Behavior**
  - Leads to compilation error

```
constexpr int divide(int n, int d) {
  return n / d;
}

constexpr auto surprise = divide(0, 0);

int main() {
  std::cout << surprise;
}
```

```
..\CTUB.cpp:7:33:   in 'constexpr' expansion of 'divide(0, 0)'
..\CTUB.cpp:4:12: error: '(0 / 0)' is not a constant expression
   return n / d;
          ~~^~~
```

Restrictions Only on Used Branch

- **If constexpr evaluation does not reach invalid statement, the code is valid**

```
constexpr void throwIfZero(int value) {
  if (value == 0) {
    throw std::logic_error{""};
  }
}

constexpr int divide(int n, int d) {
  throwIfZero(d);
  return n / d;
}

constexpr auto five = divide(120, 24);
constexpr auto failure = divide(120, 0);
```

?

```
int whatIsTheAnswer() {
  return 42;
}

static_assert(whatIsTheAnswer() == 42);
```

```
constexpr int global = 42;
constexpr int const * allocate(bool useGlobal) {
  if (useGlobal) {
    return &global;
  } else {
    return new int{};
  }
}
constexpr int const * ptr = allocate(true);
```

?

<pre>int whatIsTheAnswer() {
  return 42;
}
 	
static_assert(whatIsTheAnswer() == 42);</pre>
	Incorrect
	
	The expression in static_assert requires a compile-time expression. `whatIsTheAnswer()` is not a constexpr function.

<pre>constexpr int global = 42;
constexpr int const * allocate(bool useGlobal) {
  if (useGlobal) {
    return &global;
  } else {
    return new int{};
  }
}
constexpr int const * ptr = allocate(true);</pre>
	Correct
	
	As long as the path with the new expression is not taken, it is valid to have the code in a `constexpr` function. `allocate(false);` could not be used to initialize a compile-time constant.

Literal Types

Literal Types

- **Built-in scalar types, like `int`, `double`, pointers, enumerations, etc.**
- **Structs with some restrictions**[1]
  - Trivial destructor (non-user-defined)
  - With a `constexpr` constructor and no virtual members
- **Lambdas**
- **References**
- **Arrays of literal types**
- **`void`**
- **Literal Types can be used in `constexpr` functions, but only `constexpr` member functions can be called on values of literal type**

[1] https://en.cppreference.com/w/cpp/named_req/LiteralType

Literal Class Type Example

- **Trivial Destructor**
- **Constexpr Constructor**
  - At least one
- **Constexpr Member Functions**
  - `const` & `non-const`
  - Only `constexpr` useable in `constexpr` context
  - All are non-virtual
- **Can be a template**
- **It can still contain non-constexpr constructors and member functions**

```cpp
template <typename T>
class Vector {
  constexpr static size_t dimensions = 3;
  std::array<T, dimensions> values{};
public:
  constexpr Vector(T x, T y, T z)
    : values{x, y, z}{}
  constexpr T length() const {
    auto squares = x() * x() +
                   y() * y() +
                   z() * z();
    return std::sqrt(squares);
  }
  constexpr T & x() {
    return values[0];
  }
  constexpr T const & x() const {
  return values[0];
  }
  //...
};
```

- **Can be used in `constexpr` and non-`constexpr` contexts**
- **Non-`const` member functions can be used to modify the object**
- **`constexpr` variables are `const`**

```
constexpr Vector<double> create() {
  Vector<double> v{1.0, 1.0, 1.0};
  v.x() = 2.0;
  return v;
}

constexpr auto v = create();
static_assert(doubleEqual(v.length(), 2.4495));

int main() {
  //v.x() = 1.0;
  auto v2 = create;
  v2.x() = 2.0;
}
```

- **Note on Vector: Has hardcoded three dimensions (x/y/z).**

```
template <size_t n>
struct fact {
  static size_t const value{(n > 1)?  n * fact<n-1>::value : 1};
};

template <>
struct fact<0> { // recursion base case: template specialization
  static size_t const value = 1;
};

void testFactorialCompiletime() {
  constexpr auto result = fact<5>::value;
  ASSERT_EQUAL(result, 2 * 3 * 4 * 5);
}
```

- **"Integer" only (almost) through non-type template parameters**

- **Capture types (the types returned by lambda expressions) are literal types as well**
  - They can be used as types of `constexpr` variables
  - They can be used in `constexpr` functions
  - Restrictions to `constexpr` functions and variables apply as well
- **Examples for demonstration purposes**

```
constexpr double pi = 3.14159;

constexpr auto area = [](double r) {
  return pi * r * r;
};

constexpr auto circleArea = area(2.0);
```

```
constexpr auto cubeVolume(double x) {
  auto area = [x] {return pi * x * x;};
  return area() * x;
}

constexpr auto cV = cubeVolume(5.0);
```

Compile-Time Factorial with Variable Templates


```
template <size_t N>
constexpr size_t factorial = factorial<N - 1> * N;

template <> //Base case
constexpr size_t factorial<0> = 1;
```

- **Variable templates...**
  - ... can be `constexpr`
  - ... can be defined recursively `->` specialization to define the base case
- **Useage**
  - Template-ID (Name and template arguments)

```
static_assert(factorial<0> == 1);
static_assert(factorial<5> == 120);
```

?

```
template <std::size_t N>
constexpr size_t McCarthy91 = [] {
    if constexpr (N <= 100) {
      return McCarthy91<McCarthy91<(N + 11)>>;
    }
    return N - 10;
    }();
```

```
struct Point {
  constexpr Point(double x, double y)
    : x{x}, y{y}{}
  Point() : Point{0.0, 0.0}{}
private:
  double x;
  double y;
};

constexpr Point origin{0.0, 0.0};
```

?

```
template <std::size_t N>
constexpr size_t McCarthy91 = [] {
    if constexpr (N <= 100) {
      return McCarthy91<McCarthy91<(N + 11)>>;
    }
    return N - 10;
    }();
```

Correct

Variable templates can be initialized by lambdas, which are literal types, as long as there is a base case. With `constexpr if` we reach that base case.

```
struct Point {
  constexpr Point(double x, double y)
    : x{x}, y{y}{}
  Point() : Point{0.0, 0.0}{}
private:
  double x;
  double y;
};

constexpr Point origin{0.0, 0.0};
```

Correct

Point is a literal type. Not all constructors need to be `constexpr`. However, in compile-time constant expressions only the `constexpr` constructors can be used.

User-Defined Literals

- **Type for velocity**

```cpp
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**

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

Quite verbose

- **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 (Until C++20)**

- **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: `ternary_value<>`

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 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?**

```
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;
```

?

```
Distance operator"" _m(unsigned long long v) {
  return Distance{v};
}

auto depth = -15_m;
```

```
auto operator"" _regex(std::string value) {
  return std::regex{value};
}

bool isIdentifier(std::string testee) {
  auto pattern = "[a-zA-Z_][a-zA-Z0-9_]*"_re;
  return std::regex_match(testee, pattern);
}
```

?

`Distance operator"" _m(unsigned long long v) {`
`  return Distance{v};`
`}`
 	
`auto depth = -15_m;`
	Correct
	
	Even though the literal operator only takes an unsigned value, this works, if the `Distance` type has a unary minus operator. First the literal `15_m` is resolved then the unary minus is applied to the resulting object.

`auto operator"" _regex(std::string value) {`
`  return std::regex{value};`
`}`
 	
`bool isIdentifier(std::string testee) {`
`  auto pattern = "[a-zA-Z_][a-zA-Z0-9_]*"_re;`
`  return std::regex_match(testee, pattern);`
`}`
	Incorrect
	
	The possible overload of UDL operators are limited to a specific set of parameters. `std::string` is not a valid overload.

Summary

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Summary

- **Many computations for which the arguments are known upfront can be computed at compile time**
- **Until C++20 dynamic memory allocation is not possible**
- **All literal types can be used in constexpr contexts**
- **User defined literals help giving meaning to simple values**
- **They can be used to compute numbers at compile-time**

Self Study

Compile time code

- **Have a look at the following code on Godbolt and the binary that is generated**

```cpp
constexpr auto factorial(unsigned n) {
  auto result = 1u;
  for (auto i = 2u; i <= n; i++) {
    result *= i;
  }
  return result;
}

constexpr auto factorialOf5 = factorial(5);

int main() {
  static_assert(factorialOf5 == 120);
  return factorialOf5;
}
```

```asm
main:
  push rbp
  mov rbp, rsp
  mov eax, 120
  pop rbp
  ret
```

https://www.godbolt.org/z/suMDYp

- **No calls to the `factorial` function are present anymore**

- **Base case required for `printAll` because of the recursion**

```
void printAll() {
}

template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
  std::cout << first;
  if (sizeof...(Types)) {
    std::cout << ", ";
  }
  printAll(rest...);
}
```

- **Compile-time conditional inclusion statement**

```
void printAll() {
}

template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
  std::cout << first;
  if constexpr (sizeof...(Types)) {
    std::cout << ", ";
    printAll(rest...);
  }
}
```

- **Requires compile-time expression**

- **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 don't need a base case anymore!**