Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 8 – Compile-Time Computation

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 11.04.2019
FS2019

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Recap Week 7

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

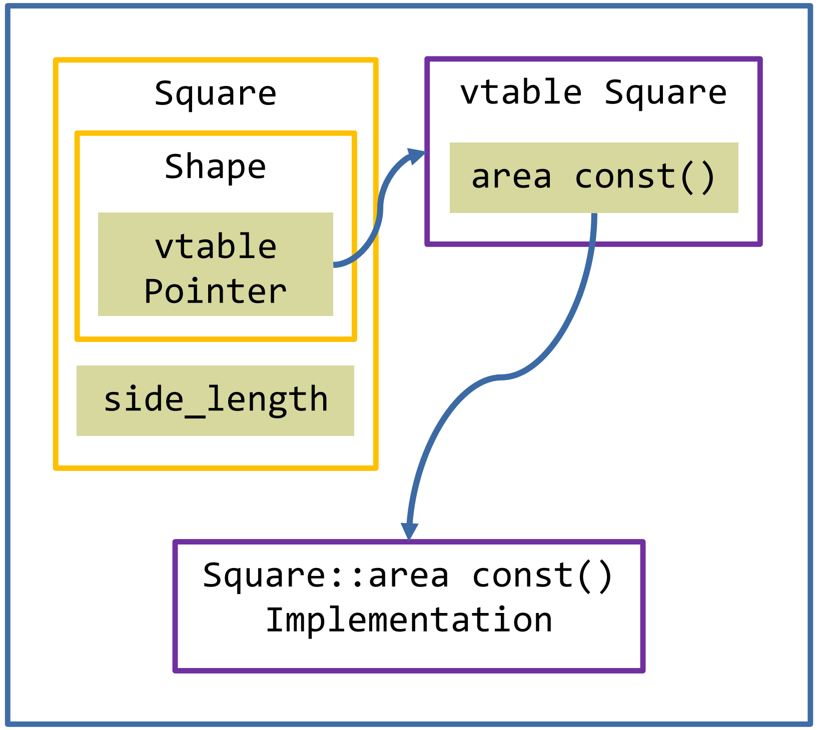
INSTITUTE FOR
SOFTWARE

- **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 const side_length;
};
```

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



- **Article on this topic:** http://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c

Tags for Dispatching: Space Ship and Drive Association


```
struct SpaceDriveTag {};

struct HyperspaceDriveTag : SpaceDriveTag {};

struct InfniteProbabilityDriveTag : SpaceDriveTag {};
```

```
struct MultiPurposeCrewVehicle;

struct GalaxyClassShip;

struct HeartOfGoldPrototype;
```

- **Approach 1: Derive space ship from the associated tag typ**
  - This is not applicable for all types (e.g. for primitive types, as we will see later)
  - This is not extensible (i.e. you cannot specify new kinds of tag kinds as a user of the API)
- **Approach 2: `SpaceshipTraits` template**

```
template<typename>
struct SpaceshipTraits {
  using Drive = SpaceDriveTag;
};
```

```
template<>
struct SpaceshipTraits<GalaxyClassShip> {
  using Drive = HyperspaceDriveTag;
};
```

Example DIY Iterator for Integers


```
struct IntIterator { /* Member Types Omitted */
  explicit IntIterator(int const start = 0) :
    value { start } {}
  bool operator==(IntIterator const & r) const {
    return value == r.value;
  }
  bool operator!=(IntIterator const & r) const {
    return !(*this == r);
  }
  value_type operator*() const {
    return value;
  }
  IntIterator & operator++() {
    ++value;
    return *this;
  }
  IntIterator operator++(int) {
    auto old = *this;
    ++(*this);
    return old;
  }
private:
  value_type value;
};
```

Explicit constructor

Implement != through operator ==

Implement postfix through prefix operators

Reuse pre-defined type

Goals

- **You can write C++ code that is evaluated by the compiler**
- **You know the restrictions of constexpr functions**
- **You can write your own literal types**
- **You know how SFINAE works and can apply it to your code**

Constexpr

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Compile-Time Computation Motivation

- **ROMable data (guaranteed!)**
  - Smaller binary
- **Better run-time performance**
- **Plain C++**
- **Not yet everything possible provided**
  - Standard library not made up for everything desirable
- **Compiletimes can increase horrendously**
  - Example that is supposed to take 24h to be compiled
    http://cpptruths.blogspot.ch/2005/11/c-templates-are-turing-complete.html

- **(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**

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

Constant expression context

Compile-time evaluation

- **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(sizeof(int) == 4, "unexpected size of int");

static_assert(isGreaterThanZero(Capacity));
```

- **`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, message);`
  - `static_assert(condition);`

```
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`**

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

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

```cpp
int local;
```

```cpp
LiteralType local{};
```

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

```cpp
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 evaluation cannot...**
  - ... 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 (until C++20)

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

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

```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);
  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);
```

?

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

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

Incorrect

The expression in static_assert requires a compile-time expression. `whatIsTheAnswer()` is not a constexpr function.

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

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.

- **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**

```
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];
  }
  //...
};
```

Using the Vector Literal Type

- **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);
```

- **Syntax**

```
template<[Parameters]>
Type name [= initialization];
```

  - Can be specialized
  - Usually `constexpr`
- **Purpose**
  - Compile-time predicates and properties of types
  - Usually applied in template meta programming
  - Before C++14 it was necessary to create a class template with a static member variable
    - Now less code is required for the same effect

```
template<typename T>
constexpr T pi = T(3.1415926535897932385);

template<typename T>
constexpr bool is_integer = false;

template<>
constexpr bool is_integer<int> = true;
```

```
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);
```

- **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!**

?

```
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};
```

Right or Wrong?

?

``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.

Compile-Time Computation (pre C++11)

- **Only preprocessor macros and templates (class & function)**
- **Templates allow Turing-complete computation**
  - But it looks ugly and can be hard to understand
  - Compiler error-messages can be lengthy
  - Compile-time long and recursion depth limited (depends on the compiler and its settings)
- **Almost only single values can be computed**
  - Arrays/structs possible, but ugly
- **Observation: static const member variables of class templates are variables that depend on template parameters**

Substitution Failure Is Not An Error (SFINAE)

Cevelop

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Overload Selection

- **What do you expect is the return value of the program on the right?**
  - None, the program does not compile!

Name increment is looked up

increment(unsinged)	increment Template

↓

Template arguments are deduced

increment(unsinged)	increment<int>(int)

↓

Best overload is selected

increment(int)

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

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

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

```
error: request for member 'increment' in 'value',
which is of non-class type 'int'
```

- **During overload resolution the template parameters in a template declaration are substituted with the deduced types**
  - This may result in template instances that cannot be compiled
  - Or otherwise suboptimal selection
- **If the substitution of template parameter fails that overload candidate is discarded**
- **Substitution failure might happen in**
  - Function return type
  - Function parameter
  - Template parameter declaration
  - And expressions in the above
- **Errors in the instance body are still errors**

Substitution

Substitution

Substitution

```
template<typename T>

T increment(T value) {
  return value.increment();
}
```

```
int increment(int value) {
  return value.increment();
} //T = int
```

Possible Solution (Not Recommended)

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

- **We can break the return type**
- **If we tell the compiler to use the type of `value.increment()` as return type for `increment<int>`**
  - That type cannot be determined during substitution

```
error: no matching function for call to 'increment(int)'
  increment(42);
              ^
note: candidate: template<class T> decltype (value.increment()) increment(T)
 auto increment(T value) -> decltype(value.increment()) {
      ^~~~~~~~~
note:   template argument deduction/substitution failed:
In substitution of 'template<class T> decltype (value.increment()) increment(T) [with T = int]':
error: request for member 'increment' in 'value', which is of non-class type 'int'
 auto increment(T value) -> decltype(value.increment()) {
```

Overload Selection with SFINAE

- **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

std::is_class

#include <type_traits>

- **Let's examine our example again**
  - We want the increment template to be selected only for class type arguments

```
template<typename T>
T increment(T value) {
  return value.increment();
}

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

- **There exists a template std::is_class<T>**
  - contains static constexpr bool value;
  - value is true if T is a class, false otherwise
- **Variable template: std::is_class_v<T>**
  - Type bool (direct access of ::value)

```
#include <type_traits>

struct S {};

int main() {
  std::is_class<S>::value; //true
  std::is_class<int>::value; //false
}
```

- **Can we apply this to the increment template directly?**
  - No, either value (true or false) is still valid
  - We need something to create an error in the type of the function

```
template<bool expr, typename T = void>
struct enable_if;
```

- **The `std::enable_if_t` template takes an expression and a type**
  - If the expression evaluates to `true` `std::enable_if_t` represents the given type
  - Otherwise it does NOT represent a type

```
int main() {
  std::enable_if_t<true, int> i;       //int
  std::enable_if_t<false, int> error; //no type
}
```

- **Inside `std::enable_if`**

```
template<bool expr,
         typename T = void>
struct enable_if {};
```

```
template<typename T>
struct enable_if<true, T> {
  using type = T;
};
```

```
template<bool expr,
         typename T = void>
using enable_if_t = typename
      enable_if<expr, T>::type;
```

```
template<typename T,  >

T  increment( T  value) {
  return value.increment();
}
```

Spots to apply enable_if (SFINAE)

- **Possibilities**

```
template<typename T>
std::enable_if_t<std::is_class<T>::value, T> increment(T value) {
  return value.increment();
}
```

```
template<typename T>
T increment(std::enable_if_t<std::is_class<T>::value, T> value) {
  return value.increment();
}
```

`enable_if` as parameter impairs type deduction

```
template<typename T, typename = std::enable_if_t<std::is_class<T>::value, void>>
T increment(T value) {
  return value.increment();
}
```

would be void per default

- **Example: Box-Container with**
  - Default constructor
  - Copy constructor
  - Move constructor
  - Size constructor

```
Box<MemoryOperationCounter> b{1};
```

```
template <typename T>
struct Box {
  Box() = default;
  Box(Box const & box)
    : items{box.items}{}
  Box(Box && box)
    : items{std::move(box.items)} {}
  explicit Box(size_t size)
    : items(size) {}
  //...
private:
  std::vector<T> items{};
};
```

- **What if we replace the copy/move constructors with a forwarding constructor?**

```
Box<MemoryOperationCounter> b{1};
```

```
template <typename T>
struct Box {
  Box() = default;
  template <typename BoxType>
  explicit Box(BoxType && other)
    : items(std::forward<BoxType>(other).items) {}
  explicit Box(size_t size)
    : items(size) {}
  //...
private:
  std::vector<T> items{};
};
```

```
Test.cpp:14:41: error: request for member 'items' in 'std::forward<int>((* & other))',
                              which is of non-class type 'int'
    : items(std::forward<BoxType>(other).items) {}
            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~
```

- **We don't want the forwarding constructor to match anything else than Boxes**
  - Type traits can be used to narrow down the valid calls.

```
template <typename T>
struct Box {
  Box() = default;
  template <typename BoxType, typename = std::enable_if_t<std::is_same_v<Box, BoxType>>>
  explicit Box(BoxType && other)
    : items(std::forward<BoxType>(other).items) {}
  explicit Box(size_t size)
    : items(size) {}
  //...
private:
  std::vector<T> items{};
};
```

`#include <type_traits>`

- **The standard library provides many predefined checks for type traits[1]**
- **A trait contains a boolean value**
- **Usually they are available in two versions**
- **Example:**
  - `std::is_same<T, U>`
  - `std::is_same_v<T, U>`

```
template <typename T, typename U>
struct is_same : false_type {
  // inherits;
  // static constexpr bool value = false;
};

template <typename T>
struct is_same<T, T> : true_type {
  // inherits;
  // static constexpr bool value = true;
};

template <typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;
```

[1] https://en.cppreference.com/w/cpp/header/type_traits

?

```
template <typename T>
std::enable_if_t<
   std::negation_v<
      std::is_reference<T>
   >
>
consume(T && value) { /*...*/ }

//This function can only be called with rvalues
```

```
template <typename T>
int convert(T value) {
  using namespace std;
  enable_if_t<
    is_constructible_v<int, T>, int
  > converted{value};
  return converted;
}
//This function can be eliminated by SFINAE
```

?

<pre>template <typename T> std::enable_if_t< std::negation_v< std::is_reference<T> > > consume(T && value) { /*...*/ } //This function can only be called with rvalues</pre>	Correct `std::is_reference` is `std::true_type` if T is a reference otherwise `std::false_type`. If the forwarding reference is initialized with an lvalue T is a reference. `std::negation_v` negates the `true/false_type`. The overload is only enabled for rvalues.
<pre>template <typename T> int convert(T value) { using namespace std; enable_if_t< is_constructible_v<int, T>, int > converted{value}; return converted; } //This function can be eliminated by SFINAE</pre>	Incorrect This is not SFINAE! It will compile if called with something that can be used to construct an int, but the overload will be taken anyway.

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**
- **SFINAE is used to eliminate overload candidates**

Self Study

Compile time code

Vanished Calls

- **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**