constexpr

In this lesson, we'll study constexpr.

WE'LL COVER THE FOLLOWING

Constant Expressions

You can define, with the keyword `constexpr`, an expression that can be evaluated at compile-time. `constexpr` can be used for variables, functions, and user-defined types. An expression that is evaluated at compile-time has a lot of advantages. A constant expression

- can be evaluated at compile-time.
- gives the compiler deep insight into the code.
- is implicitly thread-safe.

`constexpr` - Variables and Objects

If you declare a variable as `constexpr`, the compiler will evaluate them at compile-time. This holds not only true for built-in types but also for compile time. This holds not only true for built-in types but also for

instantiations of user-defined types. There are a few serious restrictions for objects in order to evaluate them at compile-time.

To make life easier for us, we will call the C types like `bool`, `char`, `int`, and `double` primitive data types. We will call the remaining data types as user-defined data types. These are for example, `std::string`, types from the C++ library and non-primitive data types. Non-primitive data types typically hold primitive data types.

Variables

By using the keyword `constexpr`, the variable becomes a constant expression.

```
constexpr double pi= 3.14;
```

Therefore, we can use the variable in contexts that require a constant expression. For example, if we want to define the size of an array. This has to be done at compile-time.

For the declaration of a `constexpr` variable, you have to keep a few rules in mind.

The variable:

- is implicitly const.
- has to be initialized.
- requires a constant expression for initialization.

The above rules make sense because if we evaluate a variable at compile-time, the variable can only depend on values that can be evaluated at compile time.

The objects are created by the invocation of the constructor and the constructor has a few special rules as well.

User-Defined Types

A `constexpr` constructor

1. can only be invoked with constant expressions.

2. cannot use exception handling.

3. has to be declared as `default` or `delete` or the function body must be empty (C++11).

The `constexpr` user-defined type

- cannot have virtual base classes.
- requires that each base object and each non-static member has to be initialized in the initialization list of the constructor or directly in the class body. Consequently, it holds that each used constructor (e.g of a base class) has to be a `constexpr` constructor and that the applied initializers have to be constant expressions.

Example

```
struct MyDouble{
  double myVal;
  constexpr MyDouble(double v): myVal(v){}
  constexpr double getVal(){return myVal;}
};
```

- The constructor has to be empty and a constant expression.
- The user-defined type can have methods which are constant expressions and cannot to be `virtual`.
- Instances of `MyDouble` can be instantiated at compile-time.

Functions

`constexpr` functions are functions that have the potential to run at compile-time. With `constexpr` functions, you can perform a lot of calculations at compile-time. Therefore, the result of the calculation is available at runtime and stored as a constant in the ROM available. In addition, `constexpr` functions are implicitly inline.

A `constexr` function can be invoked with a non-`constexpr` value. In this case, the function runs at runtime. A `constexpr` function is executed at compile-time when it is used in an expression which is evaluated at compile-time. Some examples would be when using a `static_assert` or the definition of a C-array. A `constexpr` function is also executed at compile-time, when the result is requested at compile-time, for example: `constexpr auto res = constexprFunction()`.

For `constexpr` functions there are a few restrictions:

The function

- has to be non-virtual.
- has to have arguments and a return value of a literal type. Literal types are the types of `constexpr` variables.
- can only have one return statement.
- must return a value.
- will be executed at compile-time if invoked within a constant expression.
- can only have a function body consisting of a return statement.
- must have a constant return value
- is implicitly inline.

Examples

```
constexpr int fac(int n){
    return n > 0 ? n * fac(n-1): 1;
}

constexpr int gcd(int a, int b){
  return (b==0) ? a : gcd(b, a % b);
}
```

Functions with C++14

The syntax of `constexpr` functions was massively improved with the change from C++11 to C++14. In C++11, you had to keep in mind which feature you can use in a `constexpr` functions. With C++14, you only have to keep in mind which feature you can't use in a `constexpr` function.

`constexpr` Functions in C++14

- can have variables that have to be initialized by a constant expression.

- cannot have `static` or `thread_local` data.
- can have conditional jump instructions or loop instructions.
- can have more than one instruction.

Example

```
constexpr auto gcd(int a, int b){
  while (b != 0){
    auto t= b;
    b= a % b;
    a= t;
  }
  return a;
}
```

Template Metaprogramming vs `constexpr` Functions

Characteristics	Template Metaprogramming	Constant Expressions
Execution time	Compile-time	Compile-time and runtime
Arguments	Types and values	Values
Programming paradigm	Functional	Imperative
Modification	No	Yes
Control structure	Recursion	Conditions and Loops
Conditional execution	Template specialization	Conditional statement

There are a few remarks about the above-mentioned table.

- A template metaprogram runs at compile-time, but a `constexpr` functions can run at compile-time or runtime.

- Arguments of a template (template metaprogram) can be types and values. To be more specific, a template can take types, `std::vector<int>`, values, `std::array<int, 5>`, and even templates `std::stack<int, std::vector<int>>`. `constexpr` functions are just functions which have the potential to run at compile time. Therefore, they can only accept values.

To learn more about `constexpr`, click here.

In the next lesson, we'll look at a couple of examples of `constexpr`.