Constant Expressions: constexpr

In this lesson, we will see how constexpr is used in modern C++.

WE'LL COVER THE FOLLOWING

Constant expressions

With `constexpr`, we can define 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
- can be constructed in the read-only memory (ROM-able)

constexpr - variables and objects

If we declare a variable as `constexpr`, the compiler will evaluate it at compile time. This holds true not only 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 our lives easier, we will use types built into the C++ library like `bool`, `char`, `int`, and `double`. We will call the remaining data types user-defined data types. These are all `std::string` types. User-defined types typically hold built-in types.

Variables

By using the keyword, `constexpr`, a variable can be made a constant expression.

```
constexpr double myDouble= 5.2;
```

Therefore, we can use the variable in contexts that require a constant expression, e.g., defining the size of an array. This has to be done at compile time.

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

The variable:

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

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

User-Defined types

Objects are created by the invocation of the constructor. The constructor has a few special rules.

A `constexpr` constructor:

- can only be invoked with constant expressions.
- cannot use exception handling.
- has to be declared as `default` or `delete` or the function body must be empty (C++11).

A `constexpr` user-defined type cannot have virtual base classes. It requires that each base object and each non-static member is 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 both empty and a constant expression.
- The user-defined type can have methods that may or may not be constant expressions.
- Instances of `MyDouble` can be instantiated at compile time.

Functions

`constexpr` functions are functions that have the potential to run at compile time. This means we can perform a lot of calculations at compile time, with the results available at runtime and stored as a constant in the ROM. In addition, `constexpr` functions are implicitly `inline`.

A `constexpr` 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 that is evaluated at compile time (such as `static_assert` or the definition of a C-array) or when the result is requested at compile time:

```
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. `constexpr` variables are of literal types.
- 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`.

Example

```
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, we had to remember which features we were able to use in a `constexpr` function. With C++14, we only have to remember which features we can't use in a `constexpr` function.

`constexpr` functions in C++14

- can have variables that have to be initialized by a constant expression.
- can have loops.
- cannot have `static` or `thread_local` data.
- can have conditional jump instructions or loop instructions.
- can have more than one instruction.

Example: Magic of `constexpr` functions

```
#include <iostream>

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

int main(){

 std::cout << std::endl;

  constexpr int i= gcd(11,121);

  int a= 11;
  int b= 121;
  int j= gcd(a,b);

  std::cout << "gcd(11,121): " << i << std::endl;
  std::cout << "gcd(a,b): " << j << std::endl;

  std::cout << std::endl;

}
```

The result of the variable `i` in line 16 and `j` in line 20 are calculated at compile-time and runtime respectively. The compiler would complain when we declare `j` as `constexpr`: `constexpr int j = gcd(a, b)`. The problem is that `int a`, and `int b` are not constant expressions.

The output of the program should not surprise us.

This may come as a surprise, though. Let's look at the magic with the Compiler Explorer.

```
32          call    std::basic_ostream<char, std::char_traits<char> >::operator<<(st
33          mov     DWORD PTR [rbp-4], 11
34          mov     DWORD PTR [rbp-8], 11
35          mov     DWORD PTR [rbp-12], 121
36          mov     edx, DWORD PTR [rbp-12]
37          mov     eax, DWORD PTR [rbp-8]
38          mov     esi, edx
39          mov     edi, eax
40          call    gcd(int, int)
```

```
41          mov     DWORD PTR [rbp-16], eax
```

Line 16 in the program code above boils down to the constant 11 in the following expression: `mov DWORD PTR[rbp-4], 11` (line 33 in the screenshot). In contrast, line 32 is a function call: `call gcd(int, int)` (line 40 in the screenshot).

Pure functions

We can execute `constexpr` functions at runtime. If we use the return value of a `constexpr` function in a constant expression, the compiler will perform the function at compile time. So then, is there a reason to perform a `constexpr` function at runtime? Of course! You have the `constexpr` function so we can use it at runtime. But there is another, much more convincing reason.

A `constexpr` function can potentially be performed at compile-time. There is no state at compile time, since we are in a pure functional sublanguage of the imperative programming language C++. In particular, this means that, at compile time, executed functions have to be pure functions. When we use this `constexpr` function at runtime the function stays pure.

Pure functions are functions that always return the same result when given the same arguments. Pure functions are like infinitely large tables from which we get our value. The guarantee that an expression always returns the same result when given the same arguments is called referential transparency.

Advantages

Pure functions have a lot of advantages:

- The function call can be replaced by the result.
- The execution of pure functions can automatically be distributed to other threads.
- Function calls can be reordered.
- They can easily be refactored.

The last three points hold true because pure functions have no state and therefore have no dependency on the environment. Pure functions are often called mathematical functions.

There are a lot of good reasons to use `constexpr` functions. The table shows the key points of pure and impure functions.

Pure Functions	Impure Functions
Return always the same result when given the same arguments.	May return a different result when given the same arguments.
Have never side effects.	May have side effects.
Never change the state of the program.	May change the state of the program.

Further information

- Literal type
- Compiler explorer
- Referential transparency

The examples in the next lesson will build on our understanding of this topic.