Constant Expressions: constexpr

In this lesson, we will see the usage and implementation of constexpr in Modern C++.

WE'LL COVER THE FOLLOWING

Constant Expressions

With the keyword `constexpr`, we 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 many advantages. A constant expression does the following:

- It can be evaluated at compile time.
- It gives the compiler deep insight into the code.
- It is implicitly thread-safe.
- It can be constructed in the read-only memory (ROM-able).

`constexpr` - Variables and Objects

constexpr - Variables and Objects

If we declare a variable as `constexpr`, the compiler will evaluate them at compile time. This holds true both for built-in types and for instantiations of user-defined types. There are a few restrictions for objects in order to evaluate them at compile time.

To make this process easier, we will use built-in types like `bool`, `char`, `int`, and `double`. We call the remaining data types as user-defined data types, for example, `std::string`, types from the C++ library, and user-defined data types. User-defined types typically hold built-in types.

Variables

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

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

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 must be done at compile time.

For the declaration of a `constexpr` variable, we must consider a few rules.

The variable:

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

If we evaluate a variable at compile time, the variable only depends on values that can be evaluated at compile time.

The objects are created by the invocation of the constructor.

User-Defined Types

The constructor has a few special rules.

A `constexpr` constructor

1. can only be invoked with constant expressions.

2. cannot use exception handling.
3. must 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 must 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) must be `constexpr` constructor and the applied initializers have to be constant expressions.

Example

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

- The constructor has to be an empty and a constant expression.
- The user-defined type can have methods that are constant expressions and can have methods that aren't constant expressions.
- Instances of `MyDouble` can be instantiated at compile time.

Functions

`constexpr` functions have the potential to run at compile time. With `constexpr` functions, we can perform many calculations at compile time. Therefore, the result of the calculation is available at runtime and is stored as a constant in the ROM available. 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. When used in an expression, the `constexpr` function is executed at compile time, which is evaluated at compile time such as `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 : `constexpr auto res = constexprFunction()`.

For `constexpr` functions there are a few restrictions:

The function

- must be non-virtual.
- must 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 out 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, we had to consider which features we can use in a `constexpr` function. With C++14, we need only consider which feature we cannot use in a `constexpr` function.

`constexpr` functions in C++14

- can have variables that must 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

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

Magic of `constexpr` Functions

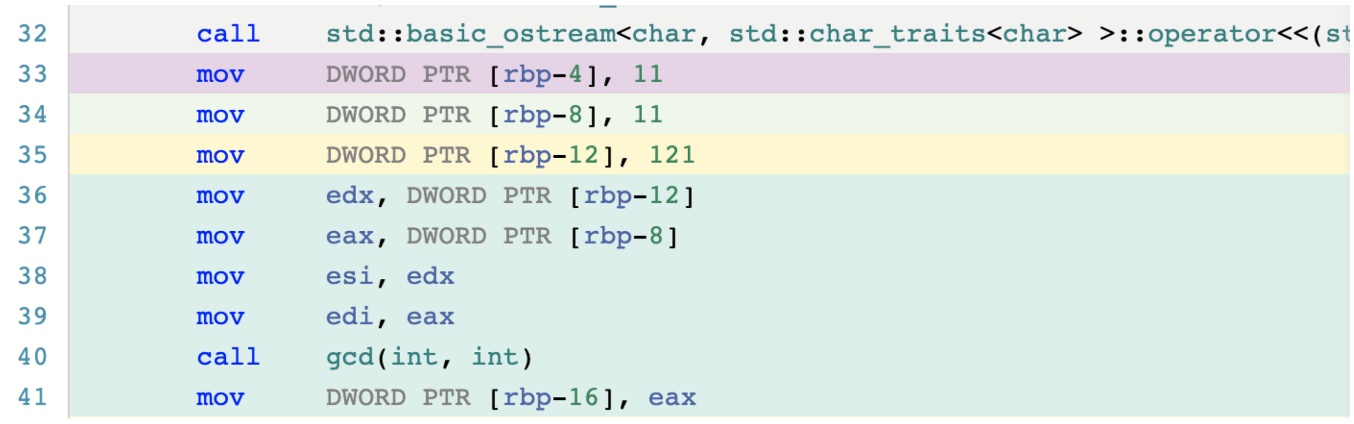
```
// constexpr14.cpp

#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);     // (1)

  int a= 11;
  int b= 121;
  int j= gcd(a,b);                  // (2)

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

  std::cout << std::endl;

}
```

Line 18 calculates the result `i` at compile time, and of line 22 `j` at runtime.

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 present us with a challenge.

The challenge may start now. Let's see the magic with 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 1 in the program code above boils down to the constant 11 in the following expression: `mov DWORD PTR[rbp-4], 11` (line 33 in 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 take the return value of a `constexpr` function by a constant expression, the compiler will perform the function at compile time. The question is: Is there a reason to perform a `constexpr` function at runtime? Let's take a look at one main reason.

A `constexpr` function can be potentially performed at compile time. There is no state at compile time, as we are in a pure functional sublanguage of the imperative programming language C++. This means that at compile time, executed functions must be pure functions. When we use the `constexpr` function at runtime, the function remains pure.

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

Pure functions are often called mathematical functions.

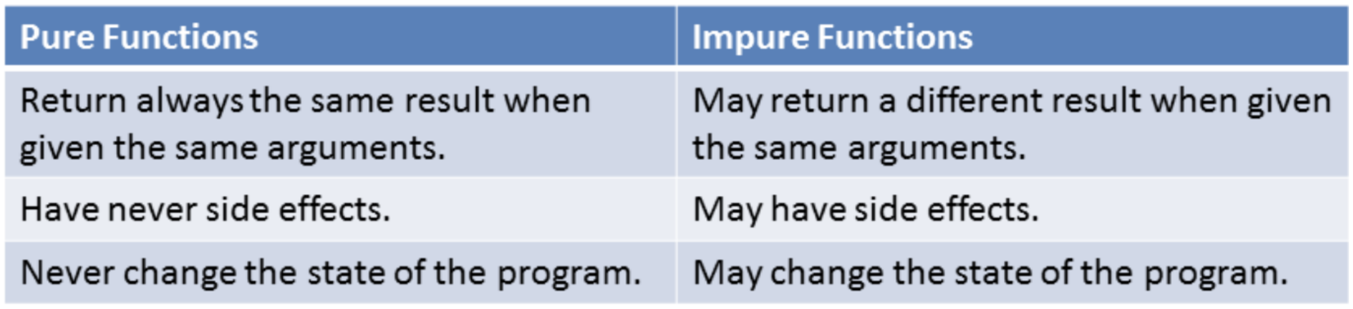
Advantages

Pure functions have many 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 because pure functions have no state and no dependency on the environment.

There are many 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.

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