constexpr (C++)

Visual Studio 2015

For the latest documentation on Visual Studio 2017, see Visual Studio 2017 Documentation.

The keyword `constexpr` was introduced in C++11 and improved in C++14. It means *constant expression*. Like `const`, it can be applied to variables so that a compiler error will be raised if any code attempts to modify the value. Unlike `const`, `constexpr` can also be applied to functions and class constructors. `constexpr` indicates that the value, or return value, is constant and, if possible, will be computed at compile time. A `constexpr` integral value can be used wherever a const integer is required, such as in template arguments and array declarations. And when a value can be computed at compile time instead of run time, it can help your program can run faster and use less memory.

Syntax

VB

```
constexpr  literal-type  identifier = constant-expression;constexpr  literal-type
identifier { constant-expression };constexpr literal-type identifier(params );constexpr ctor
(params);
```

Parameters
`params`
One or more parameters which must be a literal type (as listed below) and must itself be a constant expression.

Return Value

A constexpr variable or function must return one of the literal types, as listed below.

Literal types

To limit the complexity of computing compile time constants, and their potential impacts of compilation time, the C++14 standard requires that the types involved in constant expressions be restricted to literal types. A literal type is one whose layout can be determined at compile time. The following are the literal types:

1. void
2. scalar types
3. references

4. Arrays of void, scalar types or references

5. A class that has a trivial destructor, and one or more constexpr constructors that are not move or copy constructors. Additionally, all its non-static data members and base classes must be literal types and not volatile.

constexpr variables

The primary difference between const and constexpr variables is that the initialization of a const variable can be deferred until run time whereas a constexpr variable must be initialized at compile time. All constexpr variables are const.

```
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = exp(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

constexpr functions

A `constexpr` function is one whose return value can be computed at compile when consuming code requires it. A `constexpr` function must accept and return only literal types. When its arguments are `constexpr` values, and consuming code requires the return value at compile time, for example to initialize a `constexpr` variable or provide a non-type template argument, it produces a compile-time constant. When called with non-`constexpr` arguments, or when its value is not required at compile-time, it produces a value at run time like a regular function. (This dual behavior saves you from having to write `constexpr` and non-`constexpr` versions of the same function.)

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

Tip

Note: In the Visual Studio debugger, you can tell whether a `constexpr` function is being evaluated at compile time by putting a breakpoint inside it. If the breakpoint is hit, the function was called at run-time. If not, then the function was called at compile time.

General constexpr rules

For a function, variable, constructor or static data member to be defined as `constexpr`, it must meet certain requirements:

- A `constexpr` function can be recursive. It cannot be virtual, and its return type and parameter types must all be literal types. The body can be defined as `= default` or `= delete`. Otherwise it must follow these rules: it contains no `goto` statements, try blocks, unitialized variables, or variable definitions that are not literal types, or that are static or thread-local. Additionally, a constructor cannot be defined as constexpr if the enclosing class has any virtual base classes.
- A variable can be declared with `constexpr`, if it has a literal type and is initialized. If the initialization is performed by a constructor, the constructor must be declared as `constexpr`.
- A reference may be declared as constexpr if the object that it references has been initialized by a constant expression and any implicit conversions that are invoked during initialization are also constant expressions.
- All declarations of a `constexpr` variable or function must have the `constexpr` specifier.
- An explicit specialization of a non-constexpr template can be declared as `constexpr`:
- An explicit specialization of a `constexpr` template does not have to also be `constexpr`:
- A `constexpr` function or constructor is implicitly `inline`.

Example

The following example shows `constexpr` variables, functions and a user-defined type. Note that in the last statement in main(), the `constexpr` member function GetValue() is a run-time call because the value is not required to be known at compile time.

```
#include <iostream>

using namespace std;

// Pass by value
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};

// Pass by reference
constexpr float exp2(const float& x, const int& n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
        exp2(x * x, (n - 1) / 2) * x;
};

// Compile time computation of array length
template<typename T, int N>
constexpr int length(const T(&ary)[N])
{
    return N;
}

// Recursive constexpr function
constexpr int fac(int n)
{
    return n == 1 ? 1 : n*fac(n - 1);
}

// User-defined type
class Foo
{
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue()
    {
        return _i;
    }
private:
    int _i;
};

int main()
{
    //foo is const:
    constexpr Foo foo(5);
    // foo = Foo(6); //Error!

    //Compile time:
    constexpr float x = exp(5, 3);
    constexpr float y { exp(2, 5) };
    constexpr int val = foo.GetValue();
    constexpr int f5 = fac(5);
    const int nums[] { 1, 2, 3, 4 };
    const int nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };

    //Run time:
    cout << "The value of foo is " << foo.GetValue() << endl;

}
```

Requirements

Visual Studio 2015

See Also

Declarations and Definitions
const