50-constexpr

June 2, 2024

1 Constant expressions

A **constant expression** is an expression which can be evaluated during compilation. In C++03, it can only involve literal integers and constant variables which are themselves initialized using constant expressions. - C++11 extends the possibilities to floats, functions and objects. - C++17 adds the statement `if constexpr`, which is evaluated at compile time. - C++20 adds `consteval`, which enforces a compile-time evaluation.

1.1 `constexpr` variables

A variable declared as `constexpr` is implicitly `const`, and **must be evaluable at compile time**. Unlike a `const` variable, it should not take up any space in memory, and its value should be directly substituted in the code wherever it is used. Unlike a `#define`, it is fully pre-evaluated by the C++ compiler, and will not suffer from the pitfalls specific to preprocessor macros.

Additionally, float values are supported, and we can now initialize, within a struct|class definition, the member variables which are both `static` and`constexpr`. For example, this would not be allowed in old C++:

[3]:
```
struct X
 {
  static constexpr float pi = 3.14 ;
 } ;
```

Note 1 : the fact that the variable is eventually evaluated is up to the compiler.

Note 2 : there are situations where the compiler were already doing this optimization.

1.2 `constexpr` functions

A function declared as `constexpr` **can optionally** be evaluated at compile time, if the arguments passed to it are themselves constant expressions. Thus, the compiler can rely on such function, for example to calculate the size of a fixed array:

[10]:
```
constexpr int size( int lines, int columns )
 { return lines*columns ; }
```

[11]:
```
#include <iostream>
```

[12]:
```
constexpr int l1 = 2, c1 = 3 ;
int const l2 = 2, c2 = 3 ;
int l3 = 2, c3 = 3 ;
//...
```

[13]:
```
int s1 = size(l1,c1) ;
int s2 = size(l2,c2) ;
int s3 = size(l3,c3) ;
double a1[size(l1,c1)] = { 1., 2., 3., 4., 5., 6. } ;
double a2[size(l2,c2)] = { 1., 2., 3., 4., 5., 6. } ;
double a3[size(l3,c3)] = { 1., 2., 3., 4., 5., 6. } ;
```

```
input_line_19:7:11: error: variable-sized object may

not be initialized
double a3[size(l3,c3)] = { 1., 2., 3., 4., 5., 6. } ;
          ^~~~~~~~~~~
```

```
Interpreter Error:
```

Such functions were originally limited to a single `return` statement ! But all subsequent C++ versions have opened more and more the possibilities, so that today the main remaining prohibitions are: - the function to be `virtual`, - `try` blocks, `goto` statements, - more than one `return`statement‘, - non-literal types for parameters and return value.

1.3 constexpr constructor

By simplifying, the use of `constexpr` is allowed for a constructor if the body is empty and the members are explicitly initialized in the initialization area:

[14]:
```
class Point
 {
  public :
    constexpr Point( int a_x, int a_y ) : m_x {a_x}, m_y {a_y} {}
  private :
    int m_x, m_y ;
 } ;
```

[15]:
```
constexpr Point origin(0,0) ;
```

1.4 if constexpr (C++17)

This variant of `if` is evaluated at compile time. The condition must obviously be a constant expression, evaluable at compile time. This new *static if* is much better than a preprocessor `#if`, because its condition can be based on complex expressions evaluated by the compiler.

In this example, we are comparing two numbers, with a margin in the case of a floating point number.

[22]:
```
#include <iostream>
#include <cmath>
```

[23]:
```
template <typename T>
struct Traits { static constexpr bool is_floating_point = false ; } ;

template <>
struct Traits<float> {
  static constexpr bool is_floating_point = true ;
  static constexpr float epsilon = 1e-3 ;
 } ;

template <>
struct Traits<double> {
  static constexpr bool is_floating_point = true ;
  static constexpr double epsilon = 1e-6 ;
 } ;
```

[25]:
```
template <class T>
bool equal( T lhs, T rhs )
 {
  if constexpr (Traits<T>::is_floating_point)
   { return (std::abs(lhs-rhs)<Traits<T>::epsilon) ; }
  else
   { return lhs == rhs ; }
 }
```

[26]:
```
std::cout
  <<"1. =~ .1+.1+.1+.1+.1+.1+.1+.1+.1+.1 ? "
  <<equal(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1)
  <<std::endl ;
std::cout<<"100 =~ 10*10 ? "<<equal(100,10*10)<<std::endl ;
```

```
1. =~ .1+.1+.1+.1+.1+.1+.1+.1+.1+.1 ? 1
100 =~ 10*10 ? 1
```

To be noticed: the "if block" above is not compilable when `T` is `int`, because `Traits<int>::epsilon` does not exists. Yet, this is not a problem, since the compiler directly evaluates `Traits<T>::is_floating_point` to `false`, discards the "if block", and only compiles the "else block".

1.5 `consteval` function (C++20)

Such a function can only be called at compile time. It is meant to be an implementation which comply to the rules of constant expressions, but is not as fast as what can be done at run time.

1.6 `constinit` variable (C++20)

This weaker variant of `constexpr` let you require a compile time evaluation, although the variable is not `const`and can be modified later on. This can only be used with global or static variables, and may help to solve the *Static Initialization Order Fiasco*.

Since you safely avoid such dangerous global or static variables, this is not your concern ;)

1.7 `if consteval` (C++23)

This let you detect that the current code is evaluated at compile time, and let you use the `consteval` function optimized for this. The `else` block, on the contrary, can use the function which is optimized for run time evaluation.

2 Take away

- A `const` variable **can** be initialized at compile time.
- A `constexpr` variable **must** be initialized at compile time, and is `const`.
- A `constinit` global/static variable **must** be initialized at compile time, and is not `const`.

- `if constexpr ()` is evaluated at compile time.
- `if consteval` let you known if a compile time evaluation is under way.

- A `constexpr` function call **can** be evaluated at compile time, if its inputs are constant expressions.
- A `consteval` function call **must** be evaluated at compile time.

3 Questions ?

4 Exercise

The Fibonacci function is defined as follows: * f(0) = 0 * f(1) = 1 * f(n) = f(n-1)+f(n-2)

It is written below using old meta-programming techniques, in order to be evaluated at compile time. - Simplify it ? - Mesure separately the time of compilation and execution. - Change the type of `res` not to be `constexpr`, measure again and compare.

[27]:
```
%%file tmp.constexpr.cpp

#include <iostream>

template<int N>
struct fibonacci
 {
  enum { value = fibonacci<N-1>::value + fibonacci<N-2>::value } ;
 } ;

template<>
struct fibonacci<1>
 {
```

```
  enum { value = 1 } ;
 } ;

template<>
struct fibonacci<0>
 {
  enum { value = 0 } ;
 } ;

int main()
 {
  constexpr int res { fibonacci<36>::value } ;
  std::cout<<res<<std::endl ;
  return 0 ;
 }
```

```
Writing tmp.constexpr.cpp
```

[34]:
```
!rm -f tmp.constexpr.exe
```

[44]:
```
!bash -c "time g++ -std=c++17 tmp.constexpr.cpp -o tmp.constexpr.exe"
```

```

real    0m0.388s
user    0m0.348s
sys     0m0.039s
```

[45]:
```
!bash -c "time ./tmp.constexpr.exe"
```

```
14930352

real    0m0.006s
user    0m0.000s
sys     0m0.005s
```

4.1 Resources

- http://meetingcpp.com/blog/items/How-if-constexpr-simplifies-your-code-in-Cpp17.html
- https://www.codingame.com/playgrounds/2205/7-features-of-c17-that-will-simplify-your-code/constexpr-if
- https://linuxfr.org/news/cpp17-branche-a-la-compilation-if-constexpr
- https://solarianprogrammer.com/2017/12/27/cpp-17-constexpr-everything-as-much-as-the-compiler-can/