4-concepts

June 3, 2024

1 C++20 requirements & concepts

1.1 Motivation

Generic programming is made of functions and class templates which can be instanciated with different types. It is frequent to instantiate them with **unsuited types**, and the resulting compilation errors are generally very long and hardly understandable.

As a last resort, the template authors are providing **documentation** about the relevant parameters, and practice some tricky **template meta-programmation**.

C++20 finally brings simpler ways to define constraint on template parameters !

Among different proposals, the ISO committee has validated the flavor known as **Concepts Lite**.

1.2 Requirements and concepts in a nutshell

- A template can define **requirements** on some of its type parameters. Compiler error messages better states which parameter value does not fulfill which expected requirement.
- A typical set of requirements can be gathered in a reusable **concept**. Overload resolution takes those requirements and concepts into account.
- The standard library now provides many concepts, easy to use. Writing a new perfect concept stays an expert topic.

BEWARE : some example below may require a compiler which support concepts AND the convergence with `auto` which is associated to C++20. If using GCC >= 10, `-fconcepts` or `-fconcepts-ts` is not enough ; you should better use `-std=c++20`.

1.3 Some SFINAE example

Taking benefit from the type traits `std::is_integral` and `std::is_floating_point`, the C++17 code below is relying on SFINAE in order to implement two flavors of the `equal` function. Depending on `T`, the overload resolution will select one implementation or the other.

[19]:
```
%%file tmp.concepts.1.h

#include <iostream>
#include <type_traits>
#include <limits>
#include <cmath>
```

Overwriting tmp.concepts.1.h

[20]:
```
%%file tmp.concepts.2.h

template< typename T, std::enable_if_t<!std::is_floating_point_v<T>> * = 
 nullptr>
bool equal( T e1, T e2 )
{
  std::cout<<"(default) " ;
  return (e1==e2) ;
}

template< typename T, std::enable_if_t<std::is_floating_point_v<T>> * = nullptr>
bool equal( T e1, T e2 )
{
  std::cout<<"(floating) " ;
  return abs(e1-e2)<std::numeric_limits<T>::epsilon() ;
}
```

Overwriting tmp.concepts.2.h

[21]:
```
%%file tmp.concepts.3.h

template< typename T >
void test_equal( T v1, T v2 )
{
  std::string cmp = equal(v1,v2)?"=~":"!~" ;
  std::cout<<v1<<cmp<<v2<<std::endl ;
}
```

Overwriting tmp.concepts.3.h

[22]:
```
%%file tmp.concepts.cpp

#include "tmp.concepts.1.h"
#include "tmp.concepts.2.h"
#include "tmp.concepts.3.h"

int main()
 {
  test_equal(100,10*10) ;
  test_equal(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1) ;
 }
```

Overwriting tmp.concepts.cpp

[23]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[24]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(floating) 1=~1
```

1.4 Basic requirements

C++20 let us define many kind of requirements on the template parameters, with a syntax a lot more natural than the previous SFINAE.

[25]:
```
%%file tmp.concepts.2.h

template< typename T>
bool equal( T e1, T e2 )
{
  std::cout<<"(default) " ;
  return (e1==e2) ;
}

template< typename T>
requires
  (std::is_floating_point_v<T>) &&
  (std::numeric_limits<T>::epsilon()>0)
bool equal( T e1, T e2 )
{
  std::cout<<"(floating) " ;
  return abs(e1-e2)<std::numeric_limits<T>::epsilon() ;
}
```

```
Overwriting tmp.concepts.2.h
```

[26]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[27]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(floating) 1=~1
```

To be noticed : in the code above, I can get rid of the requirement about the default implementation of `equal`. When instanciating with a floating point type, the compiler is clever enough to understand that the second implementation meet more requirements, and is more relevant. With SFINAE, this would result into an ambiguity.

1.5 Concepts

When a given set of requirements may be reused often, one should gather them in a concept.

[28]:
```
%%file tmp.concepts.2.h

template< typename T>
bool equal( T e1, T e2 )
{
```

```
  std::cout<<"(default) " ;
  return (e1==e2) ;
}
```

```
Overwriting tmp.concepts.2.h
```

[29]:

```
%%file tmp.concepts.4.h

template< typename T>
concept MyFloatingPoint =
  (std::is_floating_point_v<T>) &&
  (std::numeric_limits<T>::epsilon()>0) ;

template<typename T>
requires MyFloatingPoint<T>
bool equal( T e1, T e2 )
{
  std::cout<<"(floating) " ;
  return abs(e1-e2)<std::numeric_limits<T>::epsilon() ;
}
```

```
Overwriting tmp.concepts.4.h
```

[30]:

```
%%file tmp.concepts.cpp

#include "tmp.concepts.1.h"
#include "tmp.concepts.2.h"
#include "tmp.concepts.3.h"
#include "tmp.concepts.4.h"

int main()
 {
  test_equal(100,10*10) ;
  test_equal(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1) ;
 }
```

```
Overwriting tmp.concepts.cpp
```

[31]:

```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[32]:

```
!./tmp.concepts.exe
```

```
(default) 100=~100
(default) 1!~1
```

1.6 Within the template head

Actually, when relevant, the concept can be directly used instead of `typename` within the template head.

[33]:
```
%%file tmp.concepts.4.h

template< typename T>
concept MyFloatingPoint =
  (std::is_floating_point_v<T>) &&
  (std::numeric_limits<T>::epsilon()>0) ;
```

```
Overwriting tmp.concepts.4.h
```

[34]:
```
%%file tmp.concepts.5.h

template<MyFloatingPoint T>
bool equal( T e1, T e2 )
{
  std::cout<<"(floating) " ;
  return abs(e1-e2)<std::numeric_limits<T>::epsilon() ;
}
```

```
Overwriting tmp.concepts.5.h
```

[35]:
```
%%file tmp.concepts.cpp

#include "tmp.concepts.1.h"
#include "tmp.concepts.2.h"
#include "tmp.concepts.3.h"
#include "tmp.concepts.4.h"
#include "tmp.concepts.5.h"

int main()
 {
  test_equal(100,10*10) ;
  test_equal(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1) ;
 }
```

```
Overwriting tmp.concepts.cpp
```

[36]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[37]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(default) 1!~1
```

1.7 With abbreviated function templates

The concepts can be used together with `auto` in the abbreviated function templates.

[38]:
```
%%file tmp.concepts.5.h

bool equal( MyFloatingPoint auto e1, MyFloatingPoint auto e2 )
```

```
{
  std::cout<<"(floating) " ;
  return abs(e1-e2)<std::numeric_limits<decltype(e1)>::epsilon() ;
}
```

```
Overwriting tmp.concepts.5.h
```

[39]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[40]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(default) 1!~1
```

To be noticed : when using the abbreviated function templates, I cannot any more enforce that `e1` and `e2` are from the same type. Consequently, when looking for the epsilon, I had to choose between the type of `e1` and the type of `e2`.

1.8 Standard concepts

Writing a bug-proof concept is actually is really expert task. Whenever you can, use the ones provided by the standard library. Not surprisingly, there is one for floating point numbers.

[41]:
```
%%file tmp.concepts.1.h

#include <iostream>
#include <type_traits>
#include <cmath>
#include <concepts>
```

```
Overwriting tmp.concepts.1.h
```

[42]:
```
%%file tmp.concepts.5.h

bool equal( std::floating_point auto e1, std::floating_point auto e2 )
{
  std::cout<<"(floating) " ;
  return abs(e1-e2)<std::numeric_limits<decltype(e1)>::epsilon() ;
}
```

```
Overwriting tmp.concepts.5.h
```

[43]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[44]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(default) 1!~1
```

1.9 With if constexpr

Concepts are usable wherever a boolean is expected, including in the condition of `if constexpr`, because they are evaluated at compile-time.

[45]:
```
%%file tmp.concepts.2.h

template< typename T>
bool equal( T e1, T e2 )
{
  if constexpr (std::floating_point<T>)
  {
    std::cout<<"(floating) " ;
    return abs(e1-e2)<std::numeric_limits<T>::epsilon() ;
  }
  else
  {
    std::cout<<"(default) " ;
    return (e1==e2) ;
  }
}
```

```
Overwriting tmp.concepts.2.h
```

[46]:
```
%%file tmp.concepts.cpp

#include "tmp.concepts.1.h"
#include "tmp.concepts.2.h"
#include "tmp.concepts.3.h"

int main()
 {
  test_equal(100,10*10) ;
  test_equal(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1) ;
 }
```

```
Overwriting tmp.concepts.cpp
```

[47]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[48]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(floating) 1=~1
```

1.10 Advanced requirements

More than the basic requirements seen before, one can use a **requires-expression** : some kind of *pseudo-fonction* which is listing expressions that must be valid.

For example, for the needs of our `test_equal` function, we may want to check that: 1. `T` is a number (integral or floating point), 2. two such numbers can be given to `equal` and get in return something that can be converted into a boolean, 3. can be sent to `std::cout`.

[162]:
```
%%file tmp.concepts.3.h

template< typename T>
concept MyComparable = requires( T v1, T v2 )
 {
  requires std::integral<T> || std::floating_point<T> ;
  { equal(v1,v2) } -> std::convertible_to<bool> ;
  std::cout<<v1<<v2 ;
 } ;

template< typename T >
requires MyComparable<T>
void test_equal( T v1, T v2 )
{
  std::string cmp = equal(v1,v2)?"=~":"!~" ;
  std::cout<<v1<<cmp<<v2<<std::endl ;
}
```

```
Overwriting tmp.concepts.3.h
```

[165]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

[164]:
```
!./tmp.concepts.exe
```

```
(default) 100=~100
(floating) 1=~1
```

1.11 Availability

GCC is probably the compiler which better supports this feature today: * GCC 6 : implements the technical specification ISO/IEC TS 19217:2015. * **GCC 10, with `-fconcepts`: implements both syntax and standard library.** * Clang 10 : implements the syntax, but not the standard library. * MSVC 19.23 : partial support of syntax and standard library.

2 Questions ?

3 Sources

- Andreas Fertig
- Cpp Reference