42-templates-novelties

June 2, 2024

1 Improvement of templates

1.1 Static assertions (C++11)

The new `static_assert` directive allows you to check a condition **during compilation** and to display the message of your choice in case of failure. It is typically used to test a type used in a template:

[ ]:
```
template <typename Int>
struct Rational {
    static_assert(sizeof(Int)>=4,"Underlying type size is not long enough\n") ;
    Int numerator ;
    Int denominator ;
} ;
```

[8]:
```
Rational<int> r1 ;
Rational<short> r2 ;
```

```
input_line_11:3:5: error: static_assert failed due to

requirement 'sizeof(short) >= 4' "Underlying type size is not long enough\n"
    static_assert(sizeof(Int)>=4,"Underlying type size is not long enough\n") ;
    ^             ~~~~~~~~~~~~~~

input_line_12:3:17: note: in instantiation of template
class '__cling_N56::Rational<short>' requested here
Rational<short> r2 ;
                ^
```

```
Interpreter Error:
```

[5]:
```
template <typename T, int size>
struct Array {
    static_assert(size>0,"Array size must be strictly positive\n") ;
    T data[size] ;
} ;
```

[6]:
```
int const n { 2 } ;
// ...
Array<int,n> a1 ;
Array<int,n-1> a2 ;
Array<int,n-2> a3 ;
```

```
input_line_11:3:5: error: static_assert failed "Array

size must be strictly positive\n"
    static_assert(size>0,"Array size must be strictly positive\n") ;
    ^              ~~~~~~

input_line_12:6:16: note: in instantiation of template
class '__cling_N56::Array<int, 0>' requested here
Array<int,n-2> a3 ;
               ^
```

```
Interpreter Error:
```

1.2 Alias templates (C++11)

It becomes possible to define some kinds of `typedef` with template parameters.

The new keyword for this is `using`.

For example, it allows to give a name to a partial specialization of a class template:

[ ]:
```
template <typename T, typename U>
class Pair { public : T x ; U y ; } ;

template <typename U>
using PairInt = Pair<int,U> ;

PairInt<double> pid ; // equivalent to Pair<int,double>
```

[ ]:
```
template <typename T>
using MapInt = std::map<int,T> ;

MapInt<double> md ; // equivalent to std::map<int,double>
```

The `using` statement should now replace`typedef` under all circumstances. It is considered more readable. Below are some examples of equivalent instructions.

[ ]:
```
#include <list>
#include <map>
#include <iostream>
```

[ ]:
```
using real = float ; // typedef float real ;

using my_map = std::map<std::vector<int>,std::list<float>> ;
using my_citr = my_map::const_iterator ;

using fptr = void(*)(int) ;
```

Starting from C ++ 14, the standard library includes some `_t` shortcuts for the templates that are used to manipulate on types. For example:

[18]:
```
template<typename T>
using remove_pointer_t = typename std::remove_pointer<T>::type ;
```

Note however that with the appearance of `auto`, it is less and less necessary to use a type alias to lighten the writing of user code.

1.3 Variadic templates (C++11)

C ++ 11 introduces the possibility of defining templates with a variable number of parameters.

Pass-through functions A simple use of `typename...` is to take a pack of arguments and transmit them to another function.

[3]:
```
#include <iostream>
#include <string>

template < typename Function, typename... Args >
void apply( std::string const & comment, Function f, Args... args )
 {
  std::cout<<"("<<comment<<") " ;
  f(args...) ;
 }
```

[4]:
```
void print( int i, double d )
 { std::cout<<i<<" "<<d<<std::endl ; }
```

[5]:
```
apply("printing",print,42,3.14) ;
```

```
(printing) 42 3.14
```

Looping on template fonction parameters When one want to loop over all the parameters, each of them with a type potentially different. With functions, you can rely on template functions which call one another recursively, and let overload resolution process:

[1]:
```
#include <iostream>

template <typename T>
void print( T last ) {
  std::cout<<last<<std::endl  ;
```

```
}

template <typename T, typename... Types>
void print( T first, Types... others ) {
  std::cout<<first<<" "  ;
  print(others...) ;
 }
```

[2]:
```
print("(printing)",42,3.14) ;
```

```
(printing) 42 3.14
```

Looping on template class parameters It gets more delicate when one want to loop the parameters of a template class. This requires a **recursive partial specialization**.

[9]:
```
%%file tmp.tuples.h

template <typename... Types> struct Tuple ;

template <typename T, typename... Types>
struct Tuple<T, Types...>
 {
  T data ;
  Tuple<Types...> others ;
 } ;

template <> struct Tuple<> {} ;
```

```
Writing tmp.tuples.h
```

[10]:
```
%%file tmp.templates.cpp

#include <iostream>
#include <string>
#include "tmp.tuples.h"

int main()
 {
  Tuple<int,double,std::string> tuple { 42, 3.14, "bonjour" } ;
  std::cout << tuple.data << std::endl ;
  std::cout << tuple.others.data << std::endl ;
  std::cout << tuple.others.others.data << std::endl ;
 }
```

```
Writing tmp.templates.cpp
```

[11]:
```
!rm -f tmp.templates.exe && g++ -std=c++17 tmp.templates.cpp -o tmp.templates.
↪exe
```

[12]:
```
!./tmp.templates.exe
```

```
42
3.14
bonjour
```

1.4 Variable templates (C++14)

C++14 introduces the possibility of making template of variables. The example usually given is a constant variable pi, that we could define with different precision for all the predefined types.

[4]:
```
%%file tmp.templates.cpp

#include <iostream>

template<typename T>
const T pi = T(3.1415926535897932385) ;

template<typename T>
T circular_area(T a_r)
 { return pi<T> * a_r * a_r ; }

int main()
 {
  std::cout.precision(18) ;
  std::cout << "double : " << circular_area(1.) << std::endl ;
  std::cout << "float  : " << circular_area(1.f) << std::endl ;
  std::cout << "int    : " << circular_area(1) << std::endl ;
 }
```

```
Writing tmp.templates.cpp
```

[5]:
```
!rm -f tmp.templates.exe && g++ -std=c++17 tmp.templates.cpp -o tmp.templates.
 ↪exe
```

[6]:
```
!./tmp.templates.exe
```

```
double : 3.14159265358979312
float  : 3.14159274101257324
int    : 3
```

Starting from C++17, the standard library includes some **_v shortcuts** for the templates that are testing type properties. For example:

[7]:
```
%%file tmp.templates.cpp

#include <type_traits>

template< class T >
bool const is_integral_v = std::is_integral<T>::value ;
```

```
template <typename IntegralT>
struct Rational {
    static_assert(is_integral_v<IntegralT>,"Bad IntegralT") ;
    IntegralT numerator ;
    IntegralT denominator ;
} ;

int main() {
  Rational<int> r1 ;
  Rational<double> r2 ;
}
```

```
Writing tmp.templates.cpp
```

[28]:
```
!rm -f tmp.templates.exe && g++ -std=c++17 tmp.templates.cpp -o tmp.templates.
↪exe
```

```
tmp.templates.cpp: In instantiation of 'struct Rational<double>':
tmp.templates.cpp:16:20:   required from here
tmp.templates.cpp:9:19: error: static assertion failed: Bad IntegralT
    9 |     static_assert(is_integral_v<IntegralT>,"Bad IntegralT") ;
      |                   ^~~~~~~~~~~~~~~~~~~~~~~~
```

1.5 Class template argument deduction (C++17)

For a template function, the compiler was already able to infer the template parameters from the type of runtime arguments. Starting with C++17, it is also okay to work with classes.

[1]:
```
template <typename Int>
struct Rational {
    Rational( Int n, Int d ) : numerator(n), denominator(d) {}
    Int numerator ;
    Int denominator ;
} ;
```

[2]:
```
Rational r { 1, 2 } ;
```

The decuction rely on implicitly-generated deduction guides. They are sometimes insufficient:

[4]:
```
%%file tmp.ctad.cpp

template <typename Int>
struct Rational {
    Int numerator ;
    Int denominator ;
} ;

int main() {
```

```
  Rational<int> r1 { 1, 2 } ;
  Rational r2 { 1, 2 } ;
}
```

```
Overwriting tmp.ctad.cpp
```

[5]:
```
!rm -f tmp.ctad.exe && g++ -std=c++17 tmp.ctad.cpp -o tmp.ctad.exe
```

```
tmp.ctad.cpp: In function ‘int main()’:
tmp.ctad.cpp:10:22: error: class template argument deduction failed:
   10 |   Rational r2 { 1, 2 } ;
      |                      ^
tmp.ctad.cpp:10:22: error: no matching function for call to ‘Rational(int, int)’
tmp.ctad.cpp:3:8: note: candidate: ‘template<class Int> Rational()->
Rational<Int>’
    3 | struct Rational {
      |        ^~~~~~~~
tmp.ctad.cpp:3:8: note:   template argument deduction/substitution failed:
tmp.ctad.cpp:10:22: note:   candidate expects 0 arguments, 2 provided
   10 |   Rational r2 { 1, 2 } ;
      |                      ^
tmp.ctad.cpp:3:8: note: candidate: ‘template<class Int>
Rational(Rational<Int>)-> Rational<Int>’
    3 | struct Rational {
      |        ^~~~~~~~
tmp.ctad.cpp:3:8: note:   template argument deduction/substitution failed:
tmp.ctad.cpp:10:22: note:   mismatched types ‘Rational<Int>’ and ‘int’
   10 |   Rational r2 { 1, 2 } ;
      |                      ^
```

In such a case, one can help the compiler by adding *user-defined deduction guides*:

[5]:
```
%%file tmp.ctad.cpp

template <typename Int>
struct Rational {
    Int numerator ;
    Int denominator ;
} ;

// user-defined deduction guide
template<class Int>
Rational(Int n, Int m) -> Rational<Int> ;

int main() {
  Rational r2 { 1, 2 } ;
}
```

```
Overwriting tmp.ctad.cpp
```

[6]:
```
!rm -f tmp.ctad.exe && g++ -std=c++17 tmp.ctad.cpp -o tmp.ctad.exe
```

1.6 Concepts (C++20)

The long awaited way to delare constraints on template parameters.

[30]:
```
%%file tmp.concepts.cpp

#include <type_traits>

template< class T >
concept integral = std::is_integral_v<T>;

template <typename Int>
requires integral<Int>
struct Rational {
    Int numerator ;
    Int denominator ;
} ;

int main() {
  Rational<double> r2 { 1., 2. } ;
}
```

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

[31]:
```
!rm -f tmp.concepts.exe && g++ -std=c++20 tmp.concepts.cpp -o tmp.concepts.exe
```

```
tmp.concepts.cpp: In function ‘int main()’:
tmp.concepts.cpp:15:18: error: template constraint failure for ‘template<class
Int>  requires  integral<Int> struct Rational’
   15 |   Rational<double> r2 { 1., 2. } ;
      |                  ^
tmp.concepts.cpp:15:18: note: constraints not satisfied
tmp.concepts.cpp: In substitution of ‘template<class Int>  requires
integral<Int> struct Rational [with Int = double]’:
tmp.concepts.cpp:15:18:   required from here
tmp.concepts.cpp:5:9:   required for the satisfaction of ‘integral<Int>’ [with
Int = double]
tmp.concepts.cpp:5:25: note: the expression ‘is_integral_v<T> [with T = double]’
evaluated to ‘false’
    5 | concept integral = std::is_integral_v<T>;
      |                    ~~~~~^~~~~~~~~~~~~~~~
tmp.concepts.cpp:15:20: error: scalar object ‘r2’ requires one element in
initializer
   15 |   Rational<double> r2 { 1., 2. } ;
      |                    ^~
```

2 Take away

- Each new version of C++ introduces some new template syntax, which is generally applied to the standard library in the following version:
  - Before, there was `std::numeric_limits<T>::is_exact`,
  - C++11 library introduced `std::is_floating_point<T>::value`,
  - C++14 syntax introduced variable templates,
  - C++17 library introduced `std::is_floating_point_v<T>`,
  - C++20 introduced concepts and `std::floating_point<T>`.
- The C++20 concepts is a real game changer, especially for the libraries authors. New libraries will now generally require C++20.

3 Questions ?

4 Exercise

Complete the `make_ptr` function, imitation of `std::make_shared`.

[ ]:
```
%%file tmp.templates.cpp

#include <memory>
#include <iostream>

class MyData
 {
  public :
    MyData( int i, double d ) : m_i {i}, m_d {d}
     { std::cout<<"MyData::MyData()"<<std::endl ; }
    int i() { return m_i ; }
    double d() { return m_d ; }
    ~MyData()
     { std::cout<<"MyData::~MyData()"<<std::endl ; }
  private :
    int m_i ;
    double m_d ;
 } ;

void display( std::shared_ptr<MyData> data_ptr )
 { std::cout<<data_ptr->i()<<" "<<data_ptr->d()<<std::endl ; }

template <typename T, typename... Args>
std::shared_ptr<T> make_ptr( Args... args )
 { return ??? ; }

int main()
 {
  auto data_ptr {make_ptr<MyData>(42,3.14)} ;
```

```
 print(data_ptr) ;
 return 0 ;
}
```

[ ]:
```
!rm -f tmp.templates.exe && g++ -std=c++17 tmp.templates.cpp -o tmp.templates.
↪exe
```

[ ]:
```
!./tmp.templates.exe
```