41-specialization-reminders

June 2, 2024

1 Template specialization

After defining a class or function template, we still can add **different implementations** for particular values of some or all of its parameters. This feature is called *template specialization*.

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

#include <cmath>
#include <limits>

template<typename T>
bool equal( T v1, T v2 )
 { return (v1==v2) ; }

template<>
bool equal( double v1, double v2 )
 { return (std::abs(v1-v2)<=std::numeric_limits<double>::epsilon()) ; }
```

```
Overwriting tmp.specialisation.h
```

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

#include <iostream>
#include "tmp.specialisation.h"

template<typename T>
void compare( T v1, T v2 )
 {
  if (equal(v1,v2)) { std::cout<<v1<<" =~ "<<v2<<std::endl ; }
  else              { std::cout<<v1<<" !~ "<<v2<<std::endl ; }
 }

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

```
Overwriting tmp.specialisation.cpp
```

[11]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++03 tmp.specialisation.cpp -o tmp.
↪specialisation.exe
```

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

```
1 =~ 1
100 =~ 100
```

1.1 Pandora's box

When specializing a class template, one can completely change its interface. This may seems a bad idea, yet it is deeply used in TMP (Template Meta Programming).

Such specialisations may stay in different files, and come into play due to a simple `#include`.

You are allowed to specialise standard library templates, such as `std::vector<MyClass>` !

Thus, in presence of templates, **the compiler becomes cautious** and will often ask the developer for additional guaranties.

1.2 Hassle with nested types, again

As we have already seen, adding the keyword `typename` is required in case of nested types to clarify that we are dealing with a type. In the below example one can suppose that it is not needed:

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

#include <iostream>
#include <vector>

template<typename Val>
void print( std::vector<Val> & col ) {
  std::vector<Val>::iterator itr ;
  for ( itr = col.begin() ; itr != col.end() ; ++itr )
    { std::cout<<(*itr)<<" " ; }
  std::cout<<std::endl ;
}

int main() {
  std::vector<int> data { 1, 2, 3, 4, 5 } ;
  print(data) ;
}
```

```
Overwriting tmp.specialisation.cpp
```

[23]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++11 tmp.specialisation.cpp -o tmp.
↪specialisation.exe
```

```
tmp.specialisation.cpp: In function ‘void print(std::vector<Val>&)’:
tmp.specialisation.cpp:7:3: error: need ‘typename’ before
‘std::vector<Val>::iterator’ because ‘std::vector<Val>’ is a dependent scope
```

```
    7 |   std::vector<Val>::iterator itr ;
      |   ^~~
tmp.specialisation.cpp:7:29: error: expected ‘;’ before ‘itr’
    7 |   std::vector<Val>::iterator itr ;
      |                             ^~~~
      |                             ;
tmp.specialisation.cpp:8:9: error: ‘itr’ was not declared in this scope
    8 |   for ( itr = col.begin() ; itr != col.end() ; ++itr )
      |         ^~~
tmp.specialisation.cpp: In instantiation of ‘void print(std::vector<Val>&) [with
Val = int]’:
tmp.specialisation.cpp:14:8:   required from here
tmp.specialisation.cpp:7:21: error: dependent-name ‘std::vector<Val>::iterator’
is parsed as a non-type, but instantiation yields a type
    7 |   std::vector<Val>::iterator itr ;
      |                     ^~~~~~~~
tmp.specialisation.cpp:7:21: note: say ‘typename std::vector<Val>::iterator’ if
a type is meant
```

Well, `std::vector` is known, but what if there exists a nasty specialisation for a certain value of `Val` which changes the meaning of `std::vector<Val>::iterator`? Something along these lines:

```
template<>
class std::vector<int>
 {
  public :
    static int iterator ;
    //...
 } ;
```

Again, the compiler needs to be reassured with the keyword `typename`. It is required for all types which are both **nested and dependent** on a template parameter, directly or indirectly.

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

#include <iostream>
#include <vector>

template<typename Val>
void print( std::vector<Val> & col ) {
  typename std::vector<Val>::iterator itr ;
  for ( itr = col.begin() ; itr != col.end() ; ++itr )
    { std::cout<<(*itr)<<" " ; }
  std::cout<<std::endl ;
}

int main() {
  std::vector<int> data { 1, 2, 3, 4, 5 } ;
  print(data) ;
```

```
}
```

```
Overwriting tmp.specialisation.cpp
```

[28]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++11 tmp.specialisation.cpp -o tmp.
↪specialisation.exe
```

[29]:
```
!./tmp.specialisation.exe
```

```
1 2 3 4 5
```

On the contrary, in all other cases the keyword `typename` is forbidden. Because of the absence of ambiguity, `typename` is also forbidden in a list of base classes, and in constructor initialization area. Generally, the compiler emits pretty clear warnings and error messages.

If a given nested and dependent type is used frequently, to avoid the drudgery of repeating `typename` everywhere, it is typical to define a `typedef` with the same name:

[33]:
```cpp
template<typename T>
struct Base
 { struct Nested { Nested( int i = 0 ) {} } ; } ;

template<typename T>
struct Derived : public Base<T>::Nested
 {
  typedef typename Base<T>::Nested Nested ;
  Derived(int x) : Nested(x)
   {
    Nested temp ;
    //.....
   }
  //.....
 } ;
```

1.3 Complication in case of inheritance

When a class inherits from a class template, the compiler suspects some possible nasty specialisation, and **does not apply the inheritance as is** !

[42]:
```cpp
%%file tmp.specialisation.cpp

#include <iostream>

template<typename Val>
struct MyContainer
 {
  int size() { return 0 ; } ;
 } ;

template<typename Val>
```

```
struct MyExtendedContainer : public MyContainer<Val>
 {
  int size_plus_one()
   { return size() + 1 ; }
 } ;

int main() {
  MyExtendedContainer<int> col {} ;
  std::cout<<col.size_plus_one()<<std::endl ;
}
```

```
Overwriting tmp.specialisation.cpp
```

[43]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++11 tmp.specialisation.cpp -o tmp.
 ↪specialisation.exe
```

```
tmp.specialisation.cpp: In member function ‘int
MyExtendedContainer<Val>::size_plus_one()’:
tmp.specialisation.cpp:14:13: error: there are no arguments to ‘size’ that
depend on a template parameter, so a declaration of ‘size’ must be available
[-fpermissive]
   14 |    { return size() + 1 ; }
      |             ^~~~
tmp.specialisation.cpp:14:13: note: (if you use ‘-fpermissive’, G++ will accept
your code, but allowing the use of an undeclared name is deprecated)
```

Three ways permit to “appease” the compiler: 1. Making the attribute visible with a help of instruction using: `... using MyContainer<Val>::size ; ....` 2. **Calling the attribute through this: `...return this->size()....`** 3. Prefixing the attribute by the class name: `...return MyContainer<Val>::size()....`

The last approach should be avoided, because it inhibits possibly virtual methods.

[44]:
```
%%file tmp.specialisation.cpp

#include <iostream>

template<typename Val>
struct MyContainer
 {
  int size() { return 0 ; } ;
 } ;

template<typename Val>
struct MyExtendedContainer : public MyContainer<Val>
 {
  int size_plus_one()
   { return this->size() + 1 ; }
 } ;
```

```
int main() {
  MyExtendedContainer<int> col {} ;
  std::cout<<col.size_plus_one()<<std::endl ;
}
```

```
Overwriting tmp.specialisation.cpp
```

[45]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++11 tmp.specialisation.cpp -o tmp.
 ↪specialisation.exe
```

[46]:
```
!./tmp.specialisation.exe
```

```
1
```

1.4 Partial specialisation

For classed with multiple parameters, it is possible to implement a **partial specialisation** on a sub-set of its parameters.

[47]:
```
%%file tmp.specialisation.h

#include <iostream>

template <typename Type, int N>
struct Array {
  Type table[N] ;
  Array() { std::cout << "General case" << std::endl ; }
} ;

template <int N>
struct Array<float, N> {
  float table[N];
  Array() { std::cout << "Type = float" << std::endl ; }
} ;

template <typename Type>
struct Array<Type, 10> {
  Type table[10] ;
  Array() { std::cout << "N = 10" << std::endl ; }
} ;

template <>
struct Array<long, 5> {
  long table[5] ;
  Array() { std::cout << "Type = long, N = 5" << std::endl ; }
} ;
```

```
Overwriting tmp.specialisation.h
```

[48]:
```
%%file tmp.specialisation.cpp

#include "tmp.specialisation.h"

typedef Array<int, 5> Array_int_5 ;  // syntactic short-cut

int main() {
    Array<double, 15> a1 ;
    Array<float, 20> a2 ;
    Array<int, 10> a3 ;
    Array<long, 5> a4 ;
    Array_int_5 a5 ;
}
```

```
Overwriting tmp.specialisation.cpp
```

[49]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++03 tmp.specialisation.cpp -o tmp.
↪specialisation.exe
```

[50]:
```
!./tmp.specialisation.exe
```

```
General case
Type = float
N = 10
Type = long, N = 5
General case
```

NOTE : partial specialisation of an integer parameter is only possible for class templates and not for function templates.

1.5 Systematic specialisation : traits

Similarly to abstract base class which only serves as a common interface (intended to be derived), one can define an empty (or almost empty) template, intended to be specialised, in order to **add some kind of properties to existing classes and/or predefined types**. We call this *traits*.

[117]:
```
%%file tmp.traits.h

template <typename T>
struct Traits {
  static const bool is_floating_point = false ;
} ;

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

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

```
Overwriting tmp.traits.h
```

[119]:
```
%%file tmp.equal.h

#include "tmp.traits.h"
#include <cmath>

template<typename T>
bool equal( T e1, T e2 )
 { return (e1==e2) ; }

template<typename T>
bool equal( float e1, float e2 )
 { return std::abs(e1-e2)<Traits<float>::epsilon ; }

template<>
bool equal( double e1, double e2 )
 { return std::abs(e1-e2)<Traits<double>::epsilon ; }
```

```
Overwriting tmp.equal.h
```

[120]:
```
%%file tmp.specialisation.cpp

#include "tmp.equal.h"
#include <iostream>

template<typename T>
void compare( T e1, T e2 )
 {
  if (equal(e1,e2)) { std::cout<<e1<<" =~ "<<e2<<std::endl ; }
  else              { std::cout<<e1<<" !~ "<<e2<<std::endl ; }
 }

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

```
Overwriting tmp.specialisation.cpp
```

[121]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++03 tmp.specialisation.cpp -o tmp.
 ↪specialisation.exe
```

[122]:
```
!./tmp.specialisation.exe
```

```
1 =~ 1
100 =~ 100
```

This idiom permits adding indirectly new members (`is_floating_point` and `epsilon`) to a class which we don't have the rights to modify, and especially predefined-type members (`float` and `double`).

1.6 Complication in case of overloading

When templates and overloaded functions (having the same name) starts to compete, the templates patrons acts frequently in an invasive way giving bringing to quite surprising results. The following code does not compile:

[132]:
```
%%file tmp.equal.h

#include "tmp.traits.h"
#include <cmath>

bool equal( unsigned e1, unsigned e2 )
 { return (e1==e2) ; }

template< typename T >
bool equal( T v1, T v2 )
 { return (std::abs(v1-v2)<Traits<T>::epsilon) ; }
```

```
Overwriting tmp.equal.h
```

[133]:
```
%%file tmp.specialisation.cpp

#include "tmp.equal.h"
#include <iostream>

template<typename T>
void compare( T e1, T e2 )
 {
  if (equal(e1,e2)) { std::cout<<e1<<" =~ "<<e2<<std::endl ; }
  else              { std::cout<<e1<<" !~ "<<e2<<std::endl ; }
 }

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

```
Overwriting tmp.specialisation.cpp
```

[134]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++03 -Wall tmp.specialisation.cpp -o␣
↪tmp.specialisation.exe
```

```
In file included from tmp.specialisation.cpp:2:
tmp.equal.h: In instantiation of ‘bool equal(T, T) [with T = int]’:
tmp.specialisation.cpp:8:12:   required from ‘void compare(T, T) [with T = int]’
tmp.specialisation.cpp:15:10:   required from here
tmp.equal.h:10:39: error: ‘epsilon’ is not a member of ‘Traits<int>’
   10 |  { return (std::abs(v1-v2)<Traits<T>::epsilon) ; }
      |                                        ^~~~~~~
```

[135]:
```
!./tmp.specialisation.exe
```

```
sh: 1: ./tmp.specialisation.exe: not found
```

When compiling `compare(100,10*10)`, because those literals are `int`, the compiler chose the template flavor of `equal`, which can better match `int` than the `equal` function using `unsigned`. Then, during instanciation of the template flavor of `equal`, it fails to find `Traits<int>::epsilon`.

[136]:
```
%%file tmp.specialisation.cpp

#include "tmp.equal.h"
#include <iostream>

template<typename T>
void compare( T e1, T e2 )
 {
  if (equal(e1,e2)) { std::cout<<e1<<" =~ "<<e2<<std::endl ; }
  else              { std::cout<<e1<<" !~ "<<e2<<std::endl ; }
 }

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

```
Overwriting tmp.specialisation.cpp
```

[137]:
```
!rm -f tmp.specialisation.exe && g++ -std=c++03 -Wall tmp.specialisation.cpp -o␣
↪tmp.specialisation.exe
```

[138]:
```
!./tmp.specialisation.exe
```

```
1 =~ 1
100 =~ 100
```

1.7 Take away

It's one of the biggest historical limitations of templates: **we have no direct way to impose constraints on parameters**. We have complicated indirect ways, within the frame of Template Meta Programming : *SFINAE*, `enable_if`... Hopefully, C++20 brings a new feature, called **concepts**, which solve this issue.

1.8 Questions ?