1-legacy-tmp

June 3, 2024

1 Legacy TMP

Template Meta Programming (TMP) means writing template code which is able to take types as input, apply operations on them, return some types as a result… at compile time.

The language was **not initially designed with this in mind**, but the specialization feature gave birth to unexpected and efficient idioms, which were harnessed and diverted very early.

It was demonstrated that TMP is "turing-complete": able to declare variables, make loops, call functions, etc. Writing **such kind of code is complicated**, the compilation time is very long, the error messages undecipherable…

Yet the libraries made this way deliver **unbeatable performance**. Below, we detail two old TMP idioms.

1.1 Make an `if` which depends on some input type

Have a look at the pseudo-code below, meant to move an iterator. We would like to support all the kinds of iterators, and benefit from a direct fast jump when this is a random access iterator.

[ ]:
```
template< typename IterT, typename DistT >
void jump( IterT & iter, DistT d )
 {
  if (iter is a random access iterator)
   {
    iter += d ;                                    // use iterator arithmetic
   }                                               // for random access iters
  else
   {
    if (d >= 0) { while (d--) ++iter ; }           // use iterative calls to
    else { while (d++) --iter ; }                  // ++ or -- for other
   }                                               // iterator categories
 }
```

If the iterator comes from the standard libray: it is defining a nested `iterator_category`. If the latter is an alias of `std::random_access_iterator_tag`, we know that the iterator has an operator `+=`. We could implement `jump()` this way:

[ ]:
```
#include <typeinfo>
```

```
template< typename IterT, typename DistT >
void jump( IterT & iter, DistT d )
 {
  if ( typeid(typename IterT::iterator_category) ==
       typeid(std::random_access_iterator_tag) )
   { iter += d ; }
  else
   {
    if (d >= 0) { while (d--) ++iter ; }
    else { while (d++) --iter ; }
   }
 }
```

This implementation suffers from two drawbacks: * the evaluation of `typeid` is done at runtime, as well as the selection of the good branch of code, which is a pitty because the compiler has all the information needed to make this choice upstream. * the code doesn't compile ! indeed, the compiler tries to compile the whole of the function regardless of the type of iterator, and will try to compile `iter+=d` even if the iterator is not a direct access iterator.

To solve these two problems, we can delegate the execution of the action to another overloaded function, to which we pass a fictitious argument of type `iterator_tag`, which allows us to select the best implementation to execute, to avoid to compile useless lines (smaller code), and to make the choice at compile time (faster code). This idiom is called "tag dispatching":

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

[ ]:
```
template< typename IterT, typename DistT >
void jump_impl
 ( IterT & iter, DistT d, std::random_access_iterator_tag )
 {
  iter += d ;
  std::cout<<"(direct jump)"<<std::endl ;
 }

template< typename IterT, typename DistT >
void jump_impl
 ( IterT & iter, DistT d, std::bidirectional_iterator_tag )
 {
  if (d >= 0) { while (d--) ++iter ; }
  else { while (d++) --iter ; }
  std::cout<<"(incremental jump)"<<std::endl ;
 }

template< typename IterT, typename DistT >
void jump( IterT & iter, DistT d )
 { jump_impl( iter, d, (typename IterT::iterator_category)() ) ; }
```

[ ]:
```
{
  std::list<int> l = { 0, 1, 2, 3, 4 } ;
  auto litr = l.begin() ;
  jump(litr,2) ;

  std::vector<int> v = { 0, 1, 2, 3, 4 } ;
  auto vitr = v.begin() ;
  jump(vitr,3) ;
}
```

This old idiom is a kind of TMP, because this is a kind of `if...else` which takes types as input, and select the relevant branch at compile time. Only the advent of C++17, and its feature `if constexpr`, has brought a simpler and more natural syntax to achieve the same result.

1.2 Make a compile time loop, thanks to recursivity

For another glance of template-based metaprogramming, let's look at compile time loops. They are made using recursion, yet not the ordinary one (a function that calls itself), but recursive template instantiations.

The "hello world" of recursivity is the calculation of factorials, illustrated below. This is the oldest traditional C++ implementation, where enums are used for static class constants.

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

[ ]:
```
template < unsigned n >
struct Factorial
 {
  static const int value = n * Factorial<n-1>::value ;
 } ;
```

[ ]:
```
template <>
struct Factorial<0>
 {
  static const int value = 1 ;
 } ;
```

[ ]:
```
std::cout << Factorial<5>::value << std::endl ;  // prints 120
std::cout << Factorial<10>::value << std::endl ; // prints 3628800
```

Recursion occurs when `Factorial<n>` refers to `Factorial<n-1>`, and so on, until a final special case stop recursion, here the specialization `Factorial<0>`. Each instance of the pattern is a structure, and each structure hijacks an `enum` to declare a `value` variable. Because we use a recursive template instantiation instead of a traditional loop, each passage of the loop creates its own structure and its own `value`.

The approach is twisted, syntactically verbose, puts the compiler on the grill... but this pre-calculates the function at the compilation stage ! C++11 then C++14 brought the `constexpr` functions, in order to make this possible in a more natural way.

1.3 Other meta-programming techniques

- Type testing and manipulation: with a high usage of specialization, the compiler can test types and apply them some modifications, such as removing a pointer, etc. The standard library is now well furnished with such utilities.
- Definition of constraints on the parameters of a template: by exploiting the previous type tests and the mechanism called SFINAE, one can restrain the allowed parameters for a given class or function template. In C++20, we hope that the "Concepts" will make this easier.
- Expression templates: for the evaluation of a mathematical expression based on operators and vector entities, one can automatically create a custom structure for this specific expression, which will only loop once at runtime, without any intermediate result.

1.4 To remember

Template-based metaprogramming is not for the average developer. It is a paradigm that arose by accident in C++, with a not very intuitive syntax, and a still insufficient support by development tools (compilers, debuggers…). However, the efficiency it brings, by moving part of the work from execution to compilation, and its ability to express certain behaviors and notations (**DSEL**) makes it a very promising tool. For library authors, it is a formidable weapon. In C++20, the introduction of **concepts** should be major facilitating step forward in this direction.

2 Questions ?

3 Exercice

Simplify the code below using `if constexpr` and modern type testing methods. Clue: instead of comparing types with `typeid`, use `std::is_same_v`.

[1]:
```
%%file tmp.legacy.cpp

#include <iostream>
#include <vector>
#include <list>

template< typename IterT, typename DistT >
void jump_impl
 ( IterT & iter, DistT d, std::random_access_iterator_tag )
 {
  iter += d ;
  std::cout<<"(direct jump)"<<std::endl ;
 }

template< typename IterT, typename DistT >
void jump_impl
 ( IterT & iter, DistT d, std::bidirectional_iterator_tag )
 {
  if (d >= 0) { while (d--) ++iter ; }
  else { while (d++) --iter ; }
```

```
  std::cout<<"(incremental jump)"<<std::endl ;
 }

template< typename IterT, typename DistT >
void jump( IterT & iter, DistT d )
 { jump_impl( iter, d, typename IterT::iterator_category() ) ; }

int main()
 {
  std::list<int> l { 0, 1, 2, 3, 4 } ;
  auto litr {l.begin()} ;
  jump(litr,2) ;
  std::cout<<(*litr)<<std::endl ;

  std::vector<int> v { 0, 1, 2, 3, 4 } ;
  auto vitr {v.begin()} ;
  jump(vitr,3) ;
  std::cout<<(*vitr)<<std::endl ;
 }
```

```
Overwriting tmp.legacy.cpp
```

[2]:
```
!rm -f tmp.legacy.exe && g++ -std=c++17 tmp.legacy.cpp -o tmp.legacy.exe
```

[3]:
```
!./tmp.legacy.exe
```

```
(incremental jump)
2
(direct jump)
3
```