40-templates-reminders

June 2, 2024

1 Templates

1.1 Principles

- Automated copy-and-paste of functions or classes, with different types and integer constants.
- Specialization: one can define a specific implementations of the function/class for specific types and/or constants.

1.1.1 Benefits

- Make code generic, yet strongly typed.
- Fully processed at compile time.
- Zero cost at run time.

1.1.2 Drawbacks

- Increased compile time.
- Long unclear error messages.
- Voluminous executables ("code bloat").

1.2 Good old-fashioned practice : factorize

Let's consider this example:

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

[2]:
```
template< typename T, int n >
struct Array
 {
  T data[n] ;
  T sum()
   {
    T res = 0 ;
    for ( int i = 0 ; i < n ; ++i )
     { res += data[i] ; }
    return res ;
   }
 } ;
```

The whole code is duplicated for any set of values for the parameters `T` and `n`:

[3]:
```
Array<int,2> a1 = { 5, -5 } ;
std::cout << a1.sum() << std::endl ;
Array<double,4> a2 = { .1, .2, .3 } ;
std::cout << a2.sum() << std::endl ;
Array<int,4> a3  = { 1, 2, 3 } ;
std::cout << a3.sum() << std::endl ;
```

```
0
0.6
6
```

One can factorize code excerpts which are independent of parameters, reducing the code bloat:

[4]:
```
template< typename T >
T sum_impl( T * data, int n )
 {
  T res = 0 ;
  for ( int i = 0 ; i < n ; ++i )
   { res += data[i] ; }
  return res ;
 }

template< typename T, int n >
struct Array
 {
  T data[n] ;
  T sum() { return sum_impl<T>(data,n) ; }
 } ;
```

[5]:
```
Array<int,2> a1 = { 5, -5 } ;
std::cout << a1.sum() << std::endl ;
Array<double,3> a2 = { .1, .2, .3 } ;
std::cout << a2.sum() << std::endl ;
Array<int,3> a3  = { 1, 2, 3 } ;
std::cout << a3.sum() << std::endl ;
```

```
0
0.6
6
```

BEWARE: factorizing reduce the executable memory footprint, but this may obstruct some compiler optimizations. For a given application and a given platform, only experimentation can decide what should be factorized.

1.3 Bonus for function: type deduction

When one call a function template, she does not need to specify explicitly the type parameters. The compiler can deduce them from the type of the function call arguments.

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

[2]:
```
struct Electron
 {
  std::string name() const { return std::string("Electron") ; }
 } ;

struct Proton
 {
  std::string name() const { return std::string("Proton") ; }
 } ;

struct Neutron
 {
  std::string name() const { return std::string("Neutron") ; }
 } ;
```

[3]:
```
template < typename Particle >
struct ParticleProxy
 {
  public :
    ParticleProxy( Particle * p ) : m_p(p) {}
    std::string name() const { return m_p->name() ; }
    ~ParticleProxy() { delete m_p ; }
  private :
    Particle * m_p ;
 } ;
```

[4]:
```
template < typename Particle >
void process( const ParticleProxy<Particle> & pp )
 { std::cout << pp.name() << std::endl ; }
```

[5]:
```
ParticleProxy<Neutron> pp(new Neutron()) ; // template parameter required
process<Neutron>(pp) ;                     // template parameter optional
```

```
Neutron
```

[6]:
```
ParticleProxy<Neutron> pp(new Neutron()) ; // template parameter required
process(pp) ;                              // template parameter deduced
```

```
Neutron
```

In order to benefit indirectly from this deduction for classes, it is rather typical to provide some kind of *make* function, which is wrapping the call to the class constructor:

[7]:
```
template < typename Particle >
ParticleProxy<Particle> make_particle_proxy( Particle * p )
 { return ParticleProxy<Particle>(p) ; }
```

[10]:
```
process(make_particle_proxy(new Neutron())) ;
```

```
Neutron
```

1.4 Hassle with implicit conversions

Since we did not make the `ParticleProxy` constructor `explicit`, implicit conversion is allowed, and the implementation of `make_particle_proxy()` can be simplified:

[11]:
```
template < typename Particle >
ParticleProxy<Particle> make_particle_proxy( Particle * p )
 { return ParticleProxy<Particle>(p) ; }
```

[12]:
```
template <typename Particle>
ParticleProxy<Particle> make_particle_proxy( Particle * p )
 { return p ; }
```

[14]:
```
process(make_particle_proxy(new Neutron())) ;
```

```
Neutron
```

We may even expect that `process` can directly transform some input parameter of type `Particle *` into the required `const ParticleProxy<Particle> &` (this is working when one provide a `char *` to a function expecting a `const std::string &`).

[15]:
```
process(new Neutron()) ;
```

```
input_line_21:2:2: error: no matching function for call

to 'process'
 process(new Neutron()) ;
 ^~~~~~~

input_line_10:2:6: note: candidate template ignored:
could not match 'ParticleProxy<type-parameter-0-0>' against 'Neutron *'
void process( const ParticleProxy<Particle> & pp )
     ^
```

```
Interpreter Error:
```

The compiler sometimes apply some irrelevant implicit conversions, and it refuses to apply this obvious one !

Actually, implicit conversions and type deductions are done at different stages of the overload resolution (search of the function that corresponds the best a given function call). **The compiler cannot do both** for a given call, and `make_particle_proxy` stay rather useful in this case

1.5 Hassle with nested types

The compiler will refuse to compile the following code:

[ ]:
```
template<typename Container>
void inspect( Container & container )
 {
  Container::iterator * x ; // DOES NOT COMPILE !
  //...
 }
```

Actually, the compiler does not know yet if `iterator` is some `Container` nested type, or some static member variable. By default, it is assuming the latter (strange choice !), and `Container::iterator * x` is seen as the multiplication of `Container::iterator` and `x`, which does not exist...

The fact that `Container::iterator` is expected to be a type can be specified with the key word `typename`:

[ ]:
```
template<typename Container>
void inspect( Container & container )
 {
  typename Container::iterator * x ;
  //...
 }
```

NOTE: surprisingly enough, we observe that the compiler try immediatly to parse and interpret the function template, without waiting it is used, and the parameters are fully known.

1.6 Take away

Despites their drawbacks, the templates bring both performance and high-level abstractions into C++. Logically, their use is still increasing in C++ 11/14/17, where the syntax is progressively improved and simplified. A really major step should be made in C++20, thanks to new features such as **modules**.

1.7 Questions ?