Object-Oriented Programming for Scientific Computing

Traits and Policies

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

Summer Semester 2018

Motivation for Traits

Problem:

- The versatile configurability of algorithms with templates often leads to the introduction of more and more template parameters.
- This makes the resulting function templates and class templates hard to read (and hard to maintain).

There are different types of template parameters:

- Indispensable template parameters
- Template parameters wich can be determined through other parameters
- Template parameters which have default values and must be specified only in very rare cases

Task: template parameters that can be deduced from other parameters should be eliminated wherever possible.

Definition of Traits

Definition: Traits

- According to the Oxford Dictionary:

  Trait a distinctive feature characterising a thing

- A definition from the field of C++ programming[a]:

  Traits represent natural additional properties of a template parameter.

[a] D. Vandevoorde, N. M. Josuttis: C++ Templates — The Complete Guide, Addison Wesley 2003

Example for Traits

Summing up a Sequence:

The sum over a set of values that are stored in a C array can be written as follows:

```
template<typename T>
T accum(T const * begin, T const * end)
{
  T result = T();
  for(; begin != end; ++begin)
    result += *begin;
  return result;
}
```

Problem:

- A problem arises when the value range of the elements isn't large enough to store the sum without overflow.

Example for Traits

```cpp
#include <iostream>
#include "accum1.h"

int main()
{
    char name[] = "templates";
    int length = sizeof(name);
    std::cout << accum(&name[0], &name[length-1])/length << std::endl;
}
```

- If `accum` is used to calculate the mean of the `char` variables in the word "templates", one receives −4 (which is neither an ASCII code nor the mean of the numerical values).
- Therefore, we need a way to specify the correct return type of `accum`.

Note: The `sizeof` trick above only works because `char` has size 1, i.e. using vectors and iterators instead of arrays and pointers is equally readable but much safer!

Example for Traits

The introduction of an additional template parameter for this special case results in code that is harder to read and use:

```
template<class V, class T>
V accum(T const * begin, T const * end)
{
  V result = V();
  for(; begin != end; ++begin)
    result += *begin;
  return result;
}

int main()
{
    char name[] = "templates";
    int length = sizeof(name);
    std::cout << accum<int>(&name[0], &name[length-1])/length << std::endl;
}
```

Example for Traits

- In C++11 and above, we could use a default template argument:

  ```
  template<class T, class V = T>
  V accum(T const * begin, T const * end)
  {
    ...
  }
  ```

  This would allow omitting the template argument `V` in the function call whenever `T` is large enough to hold the return value.
- However, we then would have to specify `T` and `V` for each call where they differ (default arguments always come last).
- A user of this function has to know when to use a type `V` that is different from `T` (and which one). The code provides no hint for possible options and a wrong choice can lead to severe bugs (!).

Type Traits

Solution: define the return type using template specialization

```
template<typename T>
struct AccumTraits
{
  typedef T AccumType;
};

template<>
struct AccumTraits<char>
{
  typedef int AccumType;
};
```

```
template<>
struct AccumTraits<short>
{
  typedef int AccumType;
};

template<>
struct AccumTraits<int>
{
  typedef long AccumType;
};
```

Note: The definition could also be written via C++11 aliases, e.g. `using AccumType = T;`, which can improve readability for larger trait classes.

Type Traits

Generic Definition of Return Type:

```
template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const * begin, T const * end)
{
  // shortcut for return type
  using AccumType = typename AccumTraits<T>::AccumType;

  AccumType result = AccumType(); // intialize to zero

  for(; begin != end; ++begin)
    result += *begin;

  return result;
}
```

Further Improvements:

- So far, we rely on the default constructor of our return type initializing the variable with zero:

```
AccumType result = AccumType();
for(; begin != end; ++begin)
  result += *begin;
return result;
```

- This works in the case of built-in number types.
- Unfortunately, there is no guarantee that this is the case in general.
- One solution is to add so-called value traits to the traits class.

Example for Value Traits

```
template<typename T>
struct AccumTraits
{
  typedef T AccumType;

  static AccumType zero()
  {
    return AccumType();
  }
};

template<>
struct AccumTraits<char>
{
  typedef int AccumType;

  static AccumType zero()
  {
    return 0;
  }
};
```

```
template<>
struct AccumTraits<short>
{
  typedef int AccumType;

  static AccumType zero()
  {
    return 0;
  }
};

template<>
struct AccumTraits<int>
{
  typedef long AccumType;

  static AccumType zero()
  {
    return 0;
  }
};
```

Example for Value Traits

We can now extract the correct return type and correct initial value from the traits class:

```
template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const * begin, T const * end)
{
  // shortcut for return type
  typedef typename AccumTraits<T>::AccumType AccumType;

  // intialize to zero
  AccumType result = AccumTraits<T>::zero();

  for(; begin != end; ++begin)
    result += *begin;

  return result;
}
```

Type Promotion

Suppose two vectors containing objects of a number type are added:

```
template<typename T>
std::vector<T> operator+(const std::vector<T>& a, const std::vector<T>& b);
```

What should the return type be when the types differ?

```
template<typename T1, typename T2>
std::vector<???> operator+(const std::vector<T1>& a, const std::vector<T2>& b);
```

e.g.:

```
std::vector<float> a;
std::vector <complex<float> > b;
std::vector<???> c = a+b;
```

Promotion Traits

The return type selection now depends on two different types.

This can also be accomplished with the help of traits classes:

```
template<typename T1, typename T2>
std::vector<typename Promotion<T1, T2>::promoted_type>
            operator+ (const std::vector<T1>&,
                       const std::vector<T2>&);
```

The promotion traits are again defined using template specialization:

```
template<typename T1, typename T2>
struct Promotion;
```

It's easy to make a partial specialization for two identical types:

```
template<typename T>
struct Promotion<T,T>
{
    typedef T promoted_type;
};
```

Promotion Traits

Other promotion traits are defined with full template specialization:

```
template<>
struct Promotion<float, complex<float> >
{
    typedef complex<float> promoted_type;
};

template<>
struct Promotion<complex<float>, float>
{
    typedef complex<float> promoted_type;
};
```

This defines `complex<float>` as the return type for the case of `float` and `complex<float>`, irrespective of order of arguments.

Promotion Traits

Using the above approach, each traits class has to be written twice. This code duplication can be avoided through the following definition:

```
template<typename T1, typename T2>
struct Promotion
{
  typedef typename Promotion<T2,T1>::promoted_type promoted_type;
}
```

This automatically flips the template arguments in cases where only one of the two versions was specified, and makes the symmetry of the promotion explicit.

The compiler keeps track of the instantiation of templates and therefore recognizes when an infinite recursion would occur (because neither version has been defined).

Promotion Traits

The function for the addition of two vectors can then be written as follows:

```
template<typename T1, typename T2>
std::vector<typename Promotion<T1,T2>::promoted_type>
operator+(const std::vector<T1>& a, const std::vector<T2>& b)
{
  using T3    = typename Promotion<T1,T2>::promoted_type;
  using Iter1 = typename std::vector<T1>::const_iterator;
  using Iter2 = typename std::vector<T2>::const_iterator;
  using Iter3 = typename std::vector<T3>::iterator;

  if (a.size() != b.size())
    throw "vectors have different size";

  std::vector<T3> c(a.size());
  Iter1 i1 = a.begin();
  Iter2 i2 = b.begin();
  Iter3 i3 = c.begin();
  for( ; i1 != a.end(); ++i1, ++i2, ++i3)
    *i3 = *i1 + *i2;
  return c;
}
```

Promotion Traits

Or in the most general form with a generic container:

```
template<typename T1, typename T2,  template<typename U,
     typename = std::allocator<U> > class Cont>
Cont<typename Promotion<T1,T2>::promoted_type>
operator+(const Cont<T1>& a, const Cont<T2>& b)
{
  using T3    = typename Promotion<T1,T2>::promoted_type;
  using Iter1 = typename Cont<T1>::const_iterator;
  using Iter2 = typename Cont<T2>::const_iterator;
  using Iter3 = typename Cont<T3>::iterator;

  if (a.size() != b.size())
    throw std::length_error("vector sizes don't match");

  Cont<T3> c(a.size());
  Iter1 i1 = a.begin();
  Iter2 i2 = b.begin();
  Iter3 i3 = c.begin();
  for( ; i1 != a.end(); ++i1, ++i2, ++i3)
    *i3 = *i1 + *i2;
  return c;
}
```

Promotion Traits

```
int main()
{
  std::vector<double> a(5,2.);
  std::vector<float>  b(5,3.);
  a = a + b;
  for (size_t i = 0; i < a.size(); ++i)
    std::cout << a[i] << std::endl;

  std::list<double> c;
  std::list<float>  d;
  for (int i = 0; i < 5; ++i)
  {
    c.push_back(i);
    d.push_back(i);
  }
  c = d + c;
  for (std::list<double>::iterator i = c.begin(); i != c.end(); ++i)
    std::cout << *i << std::endl;
}
```

Iterator Traits

STL iterators also export many of their properties using traits.

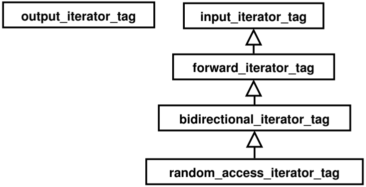
- According to the C++ standard, the following information is provided:

```
namespace std{
  template<class T>
  struct iterator_traits
  {
    typedef typename T::value_type        value_type;
    typedef typename T::difference_type   difference_type;
    typedef typename T::iterator_category iterator_category;
    typedef typename T::pointer           pointer;
    typedef typename T::reference         reference;
  };
}
```

- There is also a specialization for pointers, which are therefore a special type of iterator. For this reason generic algorithms should extract information from traits classes and not directly from the iterators.

Iterator Categories

- The category of an iterator can be queried through a tag in the iterator traits.



Example: Using the Iterator Category in Generic Code

A small example of category-aware code:

- Advance the iterator by a certain number of elements.
- If applicable, use optimized iterator functionality.

```
template<typename Iter>
void advance(Iter& pos, std::size_t dist)
{
  using IterTag = std::iterator_traits<Iter>::iterator_category;
  AdvanceHelper<Iter,IterTag>::advance(pos, dist);
}
```

Traits, Policies and the STL

As discussed earlier, the STL defines `iterator_traits` that export information about a given iterator and allow the specialization of algorithms based on present or missing capabilities.

Since C++11, it additionally defines type traits in header `<type_traits>` that can be used to specialize classes and functions based on properties of template parameters. These export either `true` or `false` as `trait_name<T>::value`.

A small selection is:

- `is_integral<T>`
- `is_floating_point<T>`
- `is_class<T>`
- `is_function<T>`
- `is_pointer<T>`
- `is_reference<T>`
- `is_polymorphic<T>`
- `is_abstract<T>`
- `is_final<T>`

There are also `common_type<T...>` (corresponds to “`Promotion<T1,T2>`”), `is_same<T,U>`, `enable_if<bool,T>`, `conditional<bool,T,F>` and (C++17) `void_t<T...>`. The last four can be used in template metaprogramming.

Traits, Policies and the STL

The STL also makes extensive use of policies:

- Each container class has at least one policy, that for allocation:

```
template<class T, class Allocator = std::allocator<T> >
class Vector;
```

- The associative containers have a comparison policy, and the unordered associative containers of C++11 have a third policy for hash generation:

```
template<class Key, class T, class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<Key,T> > >
class unordered_map;
```

- The functors and predicates (`f`, `f2`) of STL algorithms are policies that define concrete algorithm behavior:

```
template<class InputIt1, class InputIt2, class T,
       class BinaryOp1, class BinaryOp2>
T inner_product(InputIt1 first1, InputIt1 last1,
                InputIt2 first2, T value,
                BinaryOp1 op1, BinaryOp2 op2);
```

Summary

- Traits (as discussed here) can be used to determine types and values based on one or more template parameters (*extract information*).
- Policies can be used to specify parts of algorithms as template parameters (*inject behavior*).
- Combining traits and policies, a new level of abstraction can be achieved that is hard to reach in another way in C++.
- These techniques are not a part of the programming language but a convention, therefore no specific language devices are available.
- There is a second meaning of traits (not discussed here, but also very useful): template parameters that don't provide methods but collections of types and values (*inject information*).