Object-Oriented Programming for Scientific Computing

Traits and Policies

Ole Klein

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

23. Juni 2015

Motivation for Traits

Problem

- The versatile configurability of algorithms with templates often leads to the introduction of more and more template parameters.
- There are different types of template parameters:
  - Indispensable template parameters
  - Template parameters wich can be determined with the help of other template parameters
  - Template parameters which have default values and must be specified only in very rare cases

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 to be summed is not large enough to store the sum without overflow.

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 to be summed is not large enough to store the sum without overflow.

Example for Traits

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

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

- If `accum` is used to calculate the mean of the `char` variables in the word "templates", one receives -5 (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 the function `accum`.

Example for Traits

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

```
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)-1;
    std::cout << accum<int>(&name[0], &name[length])/length <<
        std::endl;
}
```

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;
};
```

Type Traits

Generic Definition of the Return Type

```
template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const *begin, T const *end)
{
  // short cut for the return type
  typedef typename AccumTraits<T>::AccumType 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 which initializes the variable with zero:

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

- Unfortunately, there is no guarantee that this is the case.
- 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;
  }
};
```

Example for Value Traits

```
template<typename T>
typename AccumTraits<T>::AccumType
         accum(T const *begin, T const *end)
{
  // short cut for the 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 of the variables in the two vectors are different?

```
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 selection of the return type now depends on two different types.
- This can also be accomplished with the aid 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> {
  public:
    typedef T promoted_type;
};
```

Promotion Traits

- Other promotion traits are defined with full template specialization:

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

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

Promotion Traits

- Since promotion traits must often be defined for many different combinations of variable types, the following macro can be helpful:

```
#define DECLARE_PROMOTE(A,B,C) \
  template<> struct Promotion<A,B> { \
    typedef C promoted_type; \
  }; \
  template<> struct Promotion<B,A> { \
    typedef C promoted_type; \
  };

DECLARE_PROMOTE(int, char, int);
DECLARE_PROMOTE(double, float, double);
DECLARE_PROMOTE(complex<float>, float, complex<float>);
// and so on...

#undef DECLARE_PROMOTE
```

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)
{
  typedef typename Promotion<T1,T2>::promoted_type T3;
  typedef typename std::vector<T3>::iterator Iterc;
  typedef typename std::vector<T1>::const_iterator Iter1;
  typedef typename std::vector<T2>::const_iterator Iter2;

  std::vector<T3> c(a.size());
  Iterc ic=c.begin();
  Iter2 i2=b.begin();
  for(Iter1 i1=a.begin();i1 != a.end(); ++i1, ++i2, ++ic)
    *ic = *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)
{
  typedef typename Promotion<T1,T2>::promoted_type T3;
  typedef typename Cont<T3>::iterator Iterc;
  typedef typename Cont<T1>::const_iterator Iter1;
  typedef typename Cont<T2>::const_iterator Iter2;

  Cont<T3> c(a.size());
  Iterc ic=c.begin();
  Iter2 i2=b.begin();
  for(Iter1 i1=a.begin();i1 != a.end(); ++i1, ++i2, ++ic)
    *ic = *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

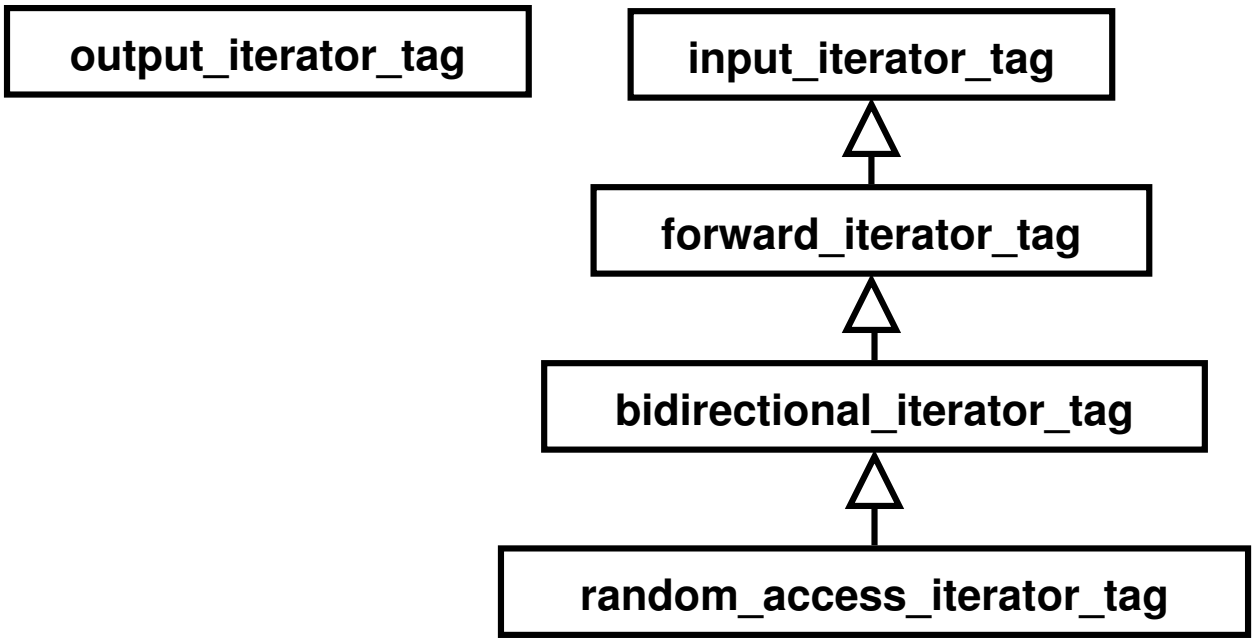
- STL iterators also export many of their properties using traits.
- According to the C++ standard, the following information about iterators is provided:

```
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.

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

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

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

Example: Using the Iterator Category in Generic Code

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

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

Example : Using the Iterator Category in Generic Code

```
#include<iterator>

template<class Iter, class IterTag>
struct AdvanceHelper{
  static void advance(Iter& pos, std::size_t dist){
    for(std::size_t i=0; i<dist; ++dist)
      ++pos;
  }
};

template<class Iter>
struct AdvanceHelper<Iter,
                     std::random_access_iterator_tag>{
  static void advance(Iter& pos, std::size_t dist){
    pos+=dist;
  }
};
```

Example: Writing HDF5 Files

```
#include <hdf5.h>

template<typename T>
struct H5ValueType
{
  static hid_t fileType()
  {
        return 0;
  };
  static hid_t memType()
  {
        return 0;
  };
};

template<>
struct H5ValueType<float>
{
  static hid_t fileType()
  {
    return H5T_IEEE_F32BE;
  }
  static hid_t memType()
  {
    return H5T_NATIVE_FLOAT;
  }
};

template<>
struct H5ValueType<double>
{
  static hid_t fileType()
  {
    return H5T_IEEE_F64BE;
  }
  static hid_t memType()
  {
    return H5T_NATIVE_DOUBLE;
  }
};
```

Example : Writing HDF5 Files

```
template<typename T>
void SaveSolutionHDF5(std::string filename, std::string fieldname,
    const std::vector<T>& outValues)
{
    ...
    hid_t dataset_id = H5Dcreate1(file_id, fieldname.c_str(),
        H5ValueType<T>::fileType(), dataspace_id, H5P_DEFAULT);

    status = H5Dwrite(dataset_id, H5ValueType<T>::memType(),
        memspace_id, dataspace_id, xferPropList, &outValues[0]);
    ...
}
```

Definition of Policies

Definition: Policies

- From the Oxford Dictionary:

  Policy any course of action adopted as advantageous or expedient

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

  Policies represent configurable behaviour for generic functions and types.

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

Extension to Other Types of Accumulation

- We have created a sum by accumulating the sequence.
- This is not the only possibility. We could also multiply the values, concatenate, or search for the maximum.
- Observation: The code of `accum` remains unchanged except for the following line:

  ```
  result += *begin
  ```

  We call this line the policy of our algorithm.
- We can put this line in a so-called policy class and pass it as a template parameter.
- A policy class is a class that provides an interface to apply one or several policies in an algorithm.

Accumulation Algorithm with Policy Class

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

  // intialize depending on policy (mult: 1, sum: 0)
  AccumType result=Policy::template init<T>();

  for(; begin!=end; ++begin)
    Policy::accumulate(result, *begin);

  return result;
}
```

Accumulation Algorithm with Policy Class

Policy for Sums

```
class SumPolicy{
public:
  template<typename T>
  static T init(){
    return AccumTraits<T>::zero();
  }

  template<typename T1, typename T2>
  static void accumulate(T1& total, T2 const& value){
    total +=value;
  }
};
```

Accumulation Algorithm with Policy Class

Policy for Multiplications

```
class MultPolicy{
public:
  template<typename T>
  static T init(){
    return AccumTraits<T>::one();
  }

  template<typename T1, typename T2>
  static void accumulate(T1& total, T2 const& value){
    total *=value;
  }
};
```

Accumulation Algorithm with Policy Class

Traits

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

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

  static AccumType one(){
    return ++AccumType();
  }

};
```

Combining Policies

- Typically, a highly configurable class will combine several different policies.
- For example, take a `SmartPointer`:

  Checking: `CheckingPolicy<T>` must provide a `check` method, which then tests whether the pointer is valid or not.

  Threading: The template class `ThreadingModel<T>` must define a `Lock` type, whose constructor gets a reference `T&` as argument.

  Storage: `StorageModel<T>` must export the types `pointer_type` and `reference_type` as well as define the methods `setPointer`, `getPointer` and `getReference`, which set the pointer of type `pointer_type`, return the pointer and return a reference of type `reference_type` to the memory location.

Combining Policies

- The user can then specify the behavior of the smart pointer class by selecting suitable template parameters:

```
typedef SmartPointer<Object,NoChecking,
                     SingleThreaded,DefaultStorage> ObjectPtr;
```

Checking Policies

```
#include<exception>

template<typename T>
struct NoChecking
{
  static void check(T)
  {}
};

template<typename T>
struct EnsureNotNull
{
  class NullPointerException : public std::exception
  {};

  static void check(const T ptr)
  {
    if(!ptr) throw NullPointerException();
  }
};
```

Default Threading Policy

```
template<typename T>
struct SingleThreaded
{
  struct Lock
  {
    Lock(const T& o)
    {}
  };
};
```

Default Storage Policy

```
template<typename T>
class DefaultStorage
{
public:
  typedef T* pointer_type;
  typedef T& reference_type;
  typedef const T* const_pointer_type;
  typedef const T& const_reference_type;
protected:
  reference_type getReference()
  {
    return *ptr_;
  }

  pointer_type getPointer()
  {
    return ptr_;
  }
  const_reference_type getReference() const
  {
    return *ptr_;
  }
```

Default Storage Policy

```
  const_pointer_type getPointer() const
  {
    return ptr_;
  }

  void setPointer(pointer_type ptr)
  {
    ptr_=ptr;
  }

  DefaultStorage() : ptr_(0)
  {}

  void releasePointer()
  {
    if (ptr_ != 0)
      delete ptr_;
  }
private:
  pointer_type ptr_;
};
```

Smart Pointer

```
template<typename T,
         template<typename> class CheckingPolicy = EnsureNotNull,
         template<typename> class ThreadingModel = SingleThreaded,
         template<typename> class StorageModel = DefaultStorage>
class SmartPointer
  : public CheckingPolicy<typename StorageModel<T>
                          ::const_pointer_type>,
    public StorageModel<T>
{
public:
  typedef typename StorageModel<T>::pointer_type pointer_type;
  typedef typename StorageModel<T>::reference_type reference_type;
  typedef typename StorageModel<T>::const_pointer_type
      const_pointer_type;
  typedef typename StorageModel<T>::const_reference_type
      const_reference_type;

  SmartPointer(pointer_type ptr){
    this->setPointer(ptr);
  }
```

Smart Pointer

```
~SmartPointer(){
  StorageModel<T>::releasePointer();
}

pointer_type operator->(){
  typename ThreadingModel<SmartPointer>::Lock guard(*this);
  CheckingPolicy<pointer_type>::check(this->getPointer());
  return this->getPointer();
}

reference_type operator*(){
  typename ThreadingModel<SmartPointer>::Lock guard(*this);
  CheckingPolicy<pointer_type>::check(this->getPointer());
  return this->getReference();
}
```

Smart Pointer

```
  const_pointer_type operator->() const{
    typename ThreadingModel<SmartPointer>::Lock guard(*this);
    CheckingPolicy<pointer_type>::check(this->getPointer());
    return this->getPointer();
  }

  const_reference_type operator*() const{
    typename ThreadingModel<SmartPointer>::Lock guard(*this);
    CheckingPolicy<pointer_type>::check(this->getPointer());
    return this->getReference();
  }
};

#endif
```

Compatible and Incompatible Policies

- Suppose we instantiate two different `SmartPointer`s:

  FastPointer: A variant without any checks
  SafePointer: A variant which checks whether the pointer is null

- Should it be possible to assign a `FastPointer` to a `SafePointer`, or vice versa?
- It is up to us whether we allow explicit conversions.
- The best and most scalable way is to initialize and copy the `SmartPointer` object *policy by policy*.

Smart Pointer

```
template<typename T,
         template<typename> class CheckingPolicy = EnsureNotNull,
         template<typename> class ThreadingModel = SingleThreaded,
         template<typename> class StorageModel = DefaultStorage>
class SmartPointer
  : public CheckingPolicy<typename StorageModel<T>
                          ::const_pointer_type>,
    public StorageModel<T>
{
public:
  template<typename T1,
           template<typename>  class CP1,
           template<typename>  class TM1,
           template<typename>  class SM1>
  SmartPointer(const SmartPointer<T1,CP1,TM1,SM1>& other) :
                       CheckingPolicy<T>(other),
                       StorageModel<T>(other)
  {}
```

Compatible and Incompatible Policies

- Suppose we want to copy an object of type `SmartPointer<ExtendedObject,NoChecking,...>` into an object of type `SmartPointer<Object,NoChecking,...>` where `ExtendedObject` is derived from `Object`,
- then the compiler tries to initialise `Object*` with a `ExtendedObject*` (which is possible) and `NoChecking` with `SmartPointer<ExtendedObject,NoChecking,...>` (also possible, since the latter class is derived from `NoChecking`).

Compatible and Incompatible Policies

- Another example: We want to copy `SmartPointer<Object,NoChecking,...>` to `SmartPointer<Object, EnforceNotNull,...>`:
- In this case the compiler is trying to hand `SmartPointer<Object,NoChecking,...>` to the constructor of `EnforceNotNull`.
- This only works if `EnforceNotNull` provides a copy constructor accepting `NoChecking` as argument.

Summary

- Traits can be used to specify types and values that depend on one or more template parameters.
- Policies can be used to specify parts of algorithms as template parameters.
- 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.