CS310

“Advanced C++:
Templates and Generic Programming”

Templates & Design
Traits and Policy Classes

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr
Traits and Policy Classes

- Templates enable us to parameterize classes and functions for various types.

- It could be tempting to introduce as many template parameters as possible to enable the customization of every aspect of a type or algorithm.

- Having to specify all the corresponding arguments in the client code is overly tedious.

- Policy classes and traits (or traits templates) are C++ programming devices that greatly facilitate the management of the sort of extra parameters that come up in the design of industrial-strength templates.
Case Study: Accumulating a Sequence

- Let's first assume that the values of the sum we want to compute are stored in an array, and we are given a pointer to the first element to be accumulated and a pointer one past the last element to be accumulated.

```cpp
template <typename T>
inline
T accum (T const* beg, T const* end)
{
    T total = T(); // assume T() actually creates a zero value.
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}
```

- The only slightly subtle decision here is how to create a zero value of the correct type to start our summation. We use the expression T() here, which normally should work for built-in numeric types like int and float.
Case Study: Accumulating a Sequence

- The code that makes use of our accum():

```cpp
int main() {
    // create array of 5 integer values.
    int num[]={1,2,3,4,5};

    // print average value.
    std::cout << "the average value of the integer values is "
             << accum(&num[0], &num[5]) / 5.
             << '\n';

    // create array of character values.
    char name[] = "templates";
    int length = sizeof(name)-1;

    // (try to) print average character value.
    std::cout << "the average value of the characters in \"".
             << name << "\" is ".
             << accum(&name[0], &name[length]) / length.
             << '\n';
}
```

- Write the code and tell me the output!!!
Case Study: Accumulating a Sequence

- The problem here is that our template was instantiated for the type `char`, which also becomes our return type, which clearly is problematic.

```cpp
template <typename T>
inline
T accum (T const* beg, T const* end)
{
    T total = T(); // assume T() actually creates a zero value.
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}
```
**Case Study: Accumulating a Sequence**

- The problem here is that our template was instantiated for the type `char`, which also becomes our return type, which clearly is problematic.

```cpp
template<typename T>
inline
T accum (T const* beg, T const* end) {
    T total = T(); // assume T() actually creates a zero value.
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}
```

- One solution could be to define the return type with another template parameter, and state it explicitly. This puts a lot of burden on the end user.
**Case Study: Accumulating a Sequence**

An alternative approach to the extra parameter is to create an association between each type T for which accum() is called and the corresponding type that should be used to hold the accumulated value. This association could be considered characteristic of the type T, and therefore the type in which the sum is computed is sometimes called a trait of T. As it turns out, our association can be encoded as specializations of a template:

```cpp
template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
   public:
      typedef int AccT;
};

template<>
class AccumulationTraits<int> {
   public:
      typedef long AccT;
};

template<>
class AccumulationTraits<float> {
   public:
      typedef double AccT;
};
```

- The template **AccumulationTraits** is called a traits template because it holds a trait of its parameter type.
Case Study: Accumulating a Sequence

• We can rewrite our accum() template as follows:

```cpp
template <typename T>:
inline:
    typename AccumulationTraits<T>::AccT accum (T const* beg,
        T const* end):
{
    // return type is traits of the element type.
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccT();  // assume T() actually creates a zero value
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}
```

• Write the code and tell me the output!!!
Case Study: Accumulating a Sequence

- We have added a very useful mechanism to customize our algorithm.
- Furthermore, if new types arise for use with accum(), an appropriate AccT can be associated with it simply by declaring an additional explicit specialization of the AccumulationTraits template.
- Note that this can be done for any type: fundamental types, types that are declared in other libraries, and so forth.
Our original accum() template uses the default constructor of the return value to initialize the result variable with what is hoped to be a zero-like value:

```
AccT total = AccT();  // assume T() actually creates a zero value
```

Clearly, there is no guarantee that this produces a good value to start the accumulation loop. Type T may not even have a default constructor.

Again, traits can come to the rescue. For our example, we can add a new value trait to our AccumulationTraits:
In this case, our new trait is a constant that can be evaluated at compile time.
**Case Study: Accumulating a Sequence**

**VALUE TRAITS**

```cpp
template <typename T>
inline
typedef typename AccumulationTraits<T>::AccT AccumulationTraits<T>::AccT;

AccT total = AccumulationTraits<T>::zero;
while (beg != end) {
    total += *beg;
    ++beg;
}
return total;
```

- A drawback of this formulation is that C++ allows us to initialize only a static constant data member inside its class if it has an integral or enumeration type. This excludes our own classes, of course, and floating-point types as well. The following specialization is, therefore, an error:

```cpp
static double const zero = 0.0;  // ERROR: not an integral type
```
Case Study: Accumulating a Sequence

VALUE TRAITS

- We prefer to implement value traits, which are not guaranteed to have integral values as inline member functions. For example, we could rewrite AccumulationTraits as follows:

```cpp
template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT;
    static AccT zero() {
      return 0;
    }
};

template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT;
    static AccT zero() {
      return 0;
    }
};
```

For the application code, the only difference is the use of function call syntax:

```cpp
AccT total = AccumulationTraits<T>::zero();
```

Clearly, traits can be more than just extra types. In our example, they can be a mechanism to provide all the necessary information that accum() needs about the element type for which it is called. **This is the key of the traits concept:** Traits provide an avenue to configure concrete elements (mostly types) for generic computations.
Case Study: Accumulating a Sequence

Policies & Policy Classes

- We have equated accumulation with summation. Clearly we can imagine other kinds of accumulations. For example, we could multiply the sequence of given values. The only `accum()` operation that needs to change is `total += *start`.

- Here is an example of how we could introduce such an interface in our Accum class template:

```cpp
template <typename T, 
    typename Policy = SumPolicy, 
    typename Traits = AccumulationTraits<T> > 
class Accum {
    public:
        typedef typename Traits::AccT AccT;
        static AccT accum ( T const* beg, T const* end ) {
            AccT total = Traits::zero();
            while (beg != end) {
                Policy::accumulate(total, *beg);
                ++beg;
            }
            return total;
        }
};
```
Case Study: Accumulating a Sequence

Policies & Policy Classes

- With this a SumPolicy could be written as follows:

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

- Write another policy called MultPolicy to multiply the elements in a sequence
Traits and Policies: What's the Difference?

- Traits represent natural additional properties of a template parameter.
- Policies represent configurable behavior for generic functions and types (often with some commonly used defaults).
Type functions

- In C and C++, functions can be called value functions: They take some values as parameters and return another value as a result.

- Now, what we have with templates are type functions: a function that takes some type arguments and produces a type or constant as a result.

- A very useful built-in type function is sizeof, which returns a constant describing the size (in bytes) of the given type argument.

```cpp
 template<typename T>
 class TypeSize {
  public:
    static size_t const value = sizeof(T);
  
 int main() {
  std::cout << "TypeSize\n::value = " << TypeSize<int>::value << std::endl;
 }
```
Determining Element Types

- Assume that we have a number of container templates such as `vector<T>`, `list<T>`, and `stack<T>`. We want a type function that, given such a container type, produces the element type. This can be achieved using partial specialization:

```cpp
template <typename T>
class ElementT;

template <typename T>
class ElementT<std::vector<T>> {
    public:
        typedef T Type;
};

template <typename T>
class ElementT<std::list<T>> {
    public:
        typedef T Type;
};

template <typename T>
void print_element_type (T const & c) {
    std::cout << "Container of "
              << typeid(Typename ElementT<T>::Type).name()
              << " elements.\n";
}
```

The use of partial specialization allows us to implement this without requiring the container types to know about the type function:

```cpp
int main() {
    std::stack<bool> s;
    print_element_type(s);
}
```
Determining Element Types

- In many cases, however, the type function is designed along with the applicable types and the implementation can be simplified. For example, if the container types define a member type `value_type` (as the standard containers do), we can write the following:

```cpp
template <typename C>
class ElementT {
    public:
        typedef typename C::value_type Type;
};
```

- How is a type function useful? It allows us to parameterize a template in terms of a container type, without also requiring parameters for the element type and other characteristics. For example, instead of:

```cpp
template <typename T, typename C>
T sum_of_elements (C const& c);
```

which requires syntax like `sum_of_elements<int>(list)` to specify the element type explicitly, we can declare:

```cpp
template<typename C>
typename ElementT<C>::Type sum_of_elements (C const& c);
```

where the element type is determined from the type function.
Determining Class Type

- With the following type function we can determine whether a type is a class type:

  ```cpp
template<typename T>
class IsClassT {
    private:
      typedef char One;
      typedef struct { char a[2]; } Two;
      template<typename C> static One test(int C::*);
      template<typename C> static Two test(...);
    public:
      enum { Yes = sizeof(IsClassT<T>::template test<T>(0)) == 1 };  
      enum { No = !Yes };  
};
```

- This template uses the SFINAE (substitution-failure-is-not-an-error) principle. The key to exploit SFINAE is to find a type construct that is invalid for function types but not for other types, or vice versa. For class types we can rely on the observation that the pointer-to-member type construct `int C::*` is valid only if `C` is a class type.

- `<type-id> (*)(<arg-list>)`

- `void foo(int (*)(double, char)); // pointer to function that takes two args`

- `void foo(int (MyClass::*)(double, char)); // pointer to a non-static member function of MyClass`

See `isClassT` for the code
Consider the following code:

```cpp
template <typename T>
void apply (T& arg, void (*func)(T))
{
    func(arg);
}

void incr (int& a)
{
    ++a;
}

void print (int a)
{
    std::cout << a << std::endl;
}

int main()
{
    int x=7;
    apply (x, print);
    apply (x, incr);
}
```

apply (x, print) is fine. With T substituted by int, the parameter types of apply() are int& and void(*)(int), which corresponds to the types of the arguments. The call apply (x, incr) is less straightforward. Matching the second parameter requires T to be substituted with int&, and this implies that the first parameter type is int& & , which ordinarily is not a legal C++ type.

See apply.cpp for the code
Tag Dispatching

- Tag dispatching is a way of using function overloading to dispatch based on properties of a type, and is often used hand in hand with traits classes.

- A good example of this synergy is the implementation of the std::advance() function in the C++ Standard Library, which increments an iterator n times.

- Depending on the kind of iterator, there are different optimizations that can be applied in the implementation.
  - If the iterator is random access, then the advance() function can simply be implemented with i += n, and is very efficient: constant time.
  - Other iterators must be advanced in steps, making the operation linear in n.
struct input_iterator_tag { };
struct bidirectional_iterator_tag { };
struct random_access_iterator_tag { };

namespace detail {
    template <class InputIterator, class Distance>
    void advance_dispatch(InputIterator& i, Distance n, input_iterator_tag) {
        while (n--) ++i;
    }
    template <class BidirectionalIterator, class Distance>
    void advance_dispatch(BidirectionalIterator& i, Distance n, bidirectional_iterator_tag) {
        if (n >= 0)
            while (n--) ++i;
        else
            while (n++) --i;
    }
    template <class RandomAccessIterator, class Distance>
    void advance_dispatch(RandomAccessIterator& i, Distance n, random_access_iterator_tag) {
        i += n;
    }
}

template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n) {
    typename iterator_traits<InputIterator>::iterator_category category;
    detail::advance_dispatch(i, n, category);
}
Problem: Design an Element library, that has the following members, and functions to manipulate these members

- Name as a string
- ID as an integer
- Address as a string

Based on zero overhead principle, I want to configure this class for any combination of those members, and I would not pay for any member or function that I do not use!

Design the code!

See the Doxygen solution to the in code documentation!