Code Reuse in Class Template Specialization

Document number: N3596
Authors: Peter Gottschling, SimuNova & TU Dresden
Date: 2013-03-15
Project: Programming Language C++, Evolution Working Group
Reply to: Peter.Gottschling@simunova.com

1 Motivation

The possibility to define specializations of class templates offers an enormous liberty unthinkable in most other languages. Techniques like enable_if would not be possible without this flexibility. However, in my (personal) experience over 90 % of class specializations duplicate over 80 % of the implementation. Conversely, the flexibility praised before is only used in few template classes and paid by code duplications in many classes. We propose a simple extension that provides full backward compatibility and avoids a lot of duplications.

2 Proposal

2.1 Generated Duplication

Very often class template specializations look like this:

```
template <typename T>
class my_class
{
    typedef T value_type;
    typedef size_t size_type;
    typedef my_class self;

    my_class(int i) : x(...), y(...) {}

    value_type f1() const {}
    value_type& f2() {}
    size_type f3() {}

    value_type x;
    size_type y;
};

template <typename U>
```

```
class my_class<std::complex<U>>
{
    typedef std::complex<U> value_type;
    typedef size_t size_type;
    typedef my_class self;

    my_class(int i) : x(...), y(...) {}

    value_type f1() const {}
    value_type& f2() {}
    size_type f3() {}

    value_type x;
    size_type y;
};
```

Possibly, we have a smart algorithm for f1 and f2 when the template argument is a complex number. The remainder of the class is the same as above after substituting T by std::complex<U>. Unfortunatlely, we have to write this down all again. Everytime, we add a new specialization to my_class we will copy the code from the original template and substitute T accordingly by hand. Needless to say that this is error-prone and a waste of time. If we add during program evolution a new function to the template we have to pay attention to update all specializations correctly.

Remark: To some extend, we can avoid the code duplication by introducing a base class and putting all common members there. In fact, this is done in practice (by me as well) but this approach quickly leads to complicated and poorly maintainable implementations. We think that the proposal here gives a much cleaner solution.

We propose that specializations can be marked as "**default**". That means that all type definitions, member variables and functions are copied from the general template (with the according substitutions). For instance:

```
template <typename T> // #1
class my_class
{
    typedef T value_type;
    typedef size_t size_type;
    typedef my_class self;

    my_class(int i) : x(...), y(...) {}

    value_type f1() const {}
    value_type& f2() {}
    size_type f3() {}

    value_type x;
    size_type y;
};

template <typename U> // #2
class my_class<std::complex<U>> = default
{
    value_type f1() const = delete;
    value_type& f2() {}
    const value_type& f4() {}
};

template <typename T> // #3
class my_class<vector<T>> = default
{
    value_type x = delete;
    size_type f3() {}
    const value_type& f4() {}
};

template <> // #4
class my_class<vector<bool>> = default
{
    const value_type& f4() {}
};
```

In the example above, the original/master class template (#1) defines a set of types, functions and variables. The specialization for complex numbers (#2) takes over most of #1's definitions (after substituting T by std::complex<U>). The function f1 is not copied (i.e. it does not exist in #2), f2 is overwritten, and f4 is added. Similarly, the specialization for vectors (#3) omits the variable x, overwrites f3, and adds f4.

The specialization for vector<bool> (#4) is a specialization of #3 and copies by default that definition. Thus, variable x does not exist in #4 either, the definition of f3 is that of #3 not that of #1, and f4 is newly defined and not taken over from #3. All other definitions are taken from #1.

2.2 Specializing Derived Classes

There can be situations where the specialized class template is derived from a class template which itself is specialized:

```
template <typename T> // #1
class base
{
    int f1() {}
};

template <> // #2
class base<float>
{
    int f1() {}
};

template <typename T> // #3
class derived
  : base<T>
{
    int f1() {}
};

template <> // #4
class derived<float> = default
{
    // which f1() ???
};
```

First, the duplication includes the inheritence. That is, derived<**float**> from base<**float**> as a consequence of the derivation #3 from #1.

This now raises the question whether #4 should inherit f1 from #2 or duplicate it from #3. The default should be that #4 duplicates f1 from #3. The duplication in the derived class can generate many members in #4 that do not exist in #2. If f1 is called from other member functions in #4 to modify member variables then the duplication is better suited to set up the objects of the derived type.

Rules like inhering from the base class if there is a specialization available otherwise duplicating would rapidly yield to confusion and further rules.[1] On the other hand, we can explicitly call the function from the base class:

```
// duplication with inheritance as above
template <> // #4
class derived<float>
{
    int f1() { return base::f1(); }
};
```

or even shorter:

```
// duplication with inheritance as above
template <> // #4
class derived<float>
{
    int f1() = inherited;
};
```

It is discussable whether this situation justifies the introduction of a new key word (which in does not invalidate existing code) or if the programmer should just call the function from the base class.

2.3 Consistent Specialization

We all know the additional programming effort when using members inherited from a dependent class template:

```
template <typename T>
class base
{
  public:
    typedef T value_type;
    value_type f1() {}
  private:
    value_type x;
};

template <typename T>
class derived
  : base<T>
{
    int f2()
```

[1]What if the derived class has a full specialization and the base class only a partial specialization? How to deal with different partial specializations that partly overlap in the base and derived class?

```
    {
        typename base::value_type y; // #1
        y= this->f1(); // #2
        this->x+= y; // #3
    }
};
```

Just as a reminder, why did we need the **typename** in line #1 and the **this->** in lines #2 and #3 again? We know that value_type is a type and that f1 is a function and that x is a variable. Well, we know this for the master template but not for all specializations and as long as we do not know the type of T in the derived and thus for the base class, we cannot say with certainty that in the corresponding specialization of base value_type is a type and x is a variable.

This leads to a question that drives me crazy since years:

Is there really one programmer out there who is insane enough to call a variable in a specialization value_type or a type f1?

I have seen a lot of template specializations and algorithms, types and variables changes a lot. Some of them might not even be needed in the specialization. But never I have seen that the same name was used for a entirely different kind of member — never ever.

The most aggressive suggestion in this regard is: just drop all "**typename**" and "**this->**" since all names that are used for types, variables, or functions in the master template will be used likewise in all specializations.

A less aggressive and much safer suggestion is to declare in the master template that all specializations will be checked for consistency.

```
template <typename T>
class [[consistent]] base
{
  public:
    typedef T value_type;
    value_type f1() {}
  private:
    value_type x;
};

template <>
class base<float>
{
    float value_type() {} // value_type must be a type
};

template <>
class base<int>
{
    typedef double* x; // x must be a variable
};
```

Obviously, the specializations above already look insane but they are legal C++ code. With the attribute [[consistent]] in the master template we can tell the compiler to look for such insanities/inconsistencies before the instantiation.

This consistency check has (at least) two important advantages:

- Many inconsistencies between master template and specializations are already detected during parsing the base class and not in the instantiation of some derived class. This early diagnosis should be much more understandable then the later problems during instantiation.
- Derived classes do not need the verbose annotations of "**typename**" and "**this**->" any longer. For each member of the base class it is known what kind it is and we can rely on the fact that all specialization have the same kind of members. Otherwise the consistency check would not have passed.

Restating the example from the beginning of the section reads now:

```
template <typename T>
class [[consistent]] base
{
  public:
    typedef T value_type;
    value_type f1() {}
  private:
    value_type x;
};

template <typename T>
class derived
  : base<T>
{
    int f2()
    {
        base::value_type y; // known to be a type
        y= f1(); // known to be a function
        x+= y; // known to be a variable
    }
};
```

From the discussion above, it seems a reasonable and desirable goal that all template classes are consistent regarding their specialization.

It is also worthwhile to distinguish between full and partial consistency. A class template is fully consistent when each specialization provides all members from the master template and they are all consistent. Partial consistency means that only a subset of the masters members are defined in the specialization but those that exist are consistent with the kind in the master.

Duplication encourages consistency but is not a guarantee.

3 Backward Compatibility

The proposed language extension should not violate backward compatibility since they are realized by new attributes and code without these attributes is handled as before. The only conflict could arise when the same attributes are already used in code.

4 Summary

The paper proposes a new feature for better code reuse by means of a user-controlled code duplication. We further address the consistency between master template and specialization which helps avoiding verbosity.

5 Acknowledgment

The author thanks Karsten Ahnert, Fabio Fracassi, Arno Schödl, and Valentin Ziegler for the discussion and different suggestions on how to express the code reuse.