Simulating Partial Specialization

Mat Marcus

Adobe Systems, Inc., 801 North 34th Street
Seattle, WA 98103, USA
mmarcus@adobe.com

Original version: September 2, 2000
Revised: October 8, 2001

Abstract

Many C++ compilers still don't support partial specialization. In such environments it is considered impossible to produce efficient easy to use container classes. We employ generative template metaprogramming to simulate partial specialization. These techniques enable more complete/and efficient STL implementations and serve as a portability aid for template metaprogramming libraries.

1 Introduction

Current C++ literature laments the lack of widespread compiler support for partial template specialization. In this paper we employ generative template metaprogramming techniques to simulate partial specialization. Our examples come from the domain of generic container libraries. We proceed from the metaprogramming perspective instead of utilizing an ad hoc collection of template tricks. That is, to quote [1],"Surprisingly, templates together with a number of other C++ features constitute a Turing-complete, compile-time sublanguage of C++. This makes C++ a two level language: A C++ program may contain both static code, which is evaluated at compile time, and dynamic code which s compiled and later executed at runtime." We will examine two representative problems that arise when implementing a generic container library for compilers without support for partial specialization.

1.1 Can't Avoid Template Bloat

Implementers of generic container libraries soon encounter the template bloat issue. Suppose we are implementing a std::vector clone called `Vector`. If we implement only one container template for all types,

```
template <typename T>
class SimpleVector;
```

then the lack of partial class pointer specialization causes template bloat. That is, the compiler will instantiate separate code for each vector of pointers even though the machine instructions are the same. This is often addressed by creating a separate container template, say

```
template <typename T>
class PtrVector : public SimpleVector<void *>
{/*Wrap base members with typecasts*/ };
```

`PtrVector`'s member functions wrap the base class members with typecasts. While this addresses the template bloat problem it introduces new problems. Now clients are inconvenienced in that they must remember to choose a different container template, `PtrVector`, if they are using pointers. Worse yet we have lost name commonality. That is, a generic function

```
template <typename T>
void foo();
```

that is instantiated for both pointer and non-pointer types cannot avoid template bloat if it uses `SimpleVector` in its body. The problem here is that we have no way to detect pointers at compile time.

1.2 Can't Recover Associated Types

Modern STL implementations can choose to optimize `std::copy` by using `std::memcpy` instead of a simple assignment loop in the body the when certain criteria are met. First the iterator types for which std::copy is instantiated must both be pointers. The underlying value types of the iterators must be the same and have trivial assignment operators. Compile time recovery of a pointer's underlying value type typically require partial specialization as with std::iterator_traits.

2 Generative Solutions

We introduce a template metafunction `IS_PTR` that detects pointer types at compile time without compiler support for partial specialization. We then leverage `IS_PTR` to address the issues raised in section §1.1-1.2. We follow the C++ template metaprogramming conventions established in [1]. In particular we will use the term metafunction to denote a struct template where the template parameters correspond to the metafunction arguments and a nested `enum`, or `typedef`, named `RET` will denote the return "value". The `IS_PTR` metafunction maps a type, `T`, to a nested enum that is accessible as

```
IS_PTR<T>::RET;//accesses return value
```

The enum will be nonzero if and only if `T` is a pointer type. That is:

```
template <typename T>
struct IS_PTR
{ RET = /*See §3 for  details*/};
```

Then, for example,

```
IS_PTR<int *>::RET == 1;
```

We also employ the remarkably useful `IF` metafunction from [1]. `IF` takes three template parameters: a `bool` condition and two types. If the `bool` is `true` the first type is returned, otherwise we return the second. That is:

```
template <bool cond, typename If_T,
  typename Else_T>
struct IF { RET = /*See Appendix for details*/};
```

Then for example:

```
IF<1+1==2,std::string,int>::RET;
```

is another name for `std::string`. Note that we use the version of `IF` that is implemented using template member functions instead of partial template specialization.

2.1 Eliminating Template Bloat

We must define one more metafunction before we can address the template bloat issue raised in §1.1, `GENERATE_VECTOR`. As its name suggests `GENERATE_VECTOR` is a `Vector` generator. It takes a type `T` and returns the appropriate container type, `PtrVector` or `SimpleVector`, according to whether `T` is a pointer type. The implementation is straightforward:

```
template <typename T>

struct GENERATE_VECTOR{

  typedef typename IF<IS_PTR<T>::RET,
        PtrVector<T>, SimpleVector<T>
    >::RET RET;
};
```

Then for example:

```
GENERATE_VECTOR<int*>::RET;
  //same as PtrVector<int*>

GENERATE_VECTOR<std::string>::RET;
  //same as SimpleVector<std::string>.
```

Now suppose a client of our container library needs a `Vector` of type `Foo`. He could write:

```
GENERATE_VECTOR<Foo>::RET v;// awkward
```

But this is rather awkward. So we introduce some syntactic sugar. We define a new struct template `Vector` that inherits from `GENERATE_VECTOR`. We must implement `Vector`'s constructors to delegate to the base constructors. This is not a problem when `PtrVector` and `SimpleVector`'s constructors have the same signature. So we have:

```
template <typename T>
struct Vector :
  public GENERATE_VECTOR<T>::Vector
{
  // Implement delegating constructors
};
```

Now at last we can write:

```
Vector<int*>  v1; // uses PtrVector<int*>

Vector<std::string> v2;
  //uses SimpleVector<std:;string>
```

without concern for template bloat just as if the compiler offered partial class pointer specialization.

2.2 std::copy: Recovering theUnderlying Type

In this section we examine the optimization criteria mentioned in §1.2: the iterator template parameters for std::copy must both be point to the same type, call it `V`, has a trivial assignment operator. We will use a narrower second criterion to simplify exposition. That is, we will only optimize if `V` is in fact plain old data (POD). We define the metafunction `IS_POD` that accepts a type and returns whether it is plain old data. That is:

```
template <typename T>
struct IS_POD {
  {enum RET = /* implementation omitted */ };
};
```

Ideally in the pointer case we would like a metafunction that could return `V`, e.g. `REMOVE_PTR`. Such a metafunction is provided by std::iterator_traits<T>::value_type on standard compilers. Then we could apply `IS_POD` to the result of `REMOVE_PTR`. Unfortunately, we know of no way to implement such a metafunction in our environment. But if we look more carefully we see that in many cases we don't actually need `V`. All we need to ask a question of V: "is `V` plain old data?" It turns out that we can devise such a metafunction! That is we can define:

```
template <typename T>
struct IS_PTR_TO_POD {
  {enum RET = /* see §3 */ };
};
```

Finally we outline an optimizing std::copy implementation below. A more complete definition is given in the Appendix. We make use of IS_PTR_TO_SAME_TYPE that returns the value of IS_SAME_TYPE on the underlying values. Also note that this technique works just as well for std::copy_backward and std::uninitialized_fill.

```
// IS_SAME_TYPE returns true when
// T and U are the same type

template <typename T, typename U>
    struct IS_SAME_TYPE {
      enum {RET = /* See §3 for details */};
    };

// Returns true if T and U are both pointers
to the same underlying value type

template <typename T, typename U>
    struct IS_PTR_TO_SAME_TYPE {
      enum {RET = /* See §3 for details */};
    };

// FastCopy and SafeCopy:
// The copy member does the real work
// See Appendix for implementation

template <typename InputIter, typename
ForwardIter>
struct SafeCopy {
    static ForwardIter copy(
      InputIter first, InputIter last,
      ForwardIter result);
    // could also declare copy_backward
    // and unitinitialized_fill here
};

template <typename InputIter, typename
ForwardIter>
```

```
struct FastCopy {
    static ForwardIter copy(
       InputIter first, InputIter last,
ForwardIter result);
    // could also declare copy_backward
    // and unitinitialized_fill here
};

/* Optimizing copy: if both iterator types
point to the same plain old data type
then use can use FastCopy whichrelies on
std::memcpy.Otherwise use SafeCopy which uses
an assignment loop.*/

template <typename InputIter, typename
ForwardIter>
inline ForwardIter copy(
  InputIter first, InputIter last,
  ForwardIter result)
{
  return IF<
      IS_PTR_TO_POD<InputIter>::RET
      && IS_PTR_TO_POD<ForwardIter>::RET
      && IS_PTR_TO_SAME_TYPE<InputIter,
          ForwardIter>::RET,
      FastCopy<InputIter, ForwardIter>,
      SafeCopy<InputIter, ForwardIter>
  >::RET::copy(first, last, result);
 }
```

3 Simulating Partial Specialization

When a compiler supports partial specialization it is not difficult to implement metafunctions such as IS_PTR. In the language of template metaprogramming partial template specialization acts as our conditional construct. So we can write something like:

```
template <typename T>
struct IS_PTR { enum RET = 0 };

template <typename T>
struct IS_PTR <T*> { enum RET = 1 };
```

We will "simulate partial specialization" to produce such metafunctions without partial specialization. Since the compiler won't discriminate templates based on a subset of types (e.g. T*) we must find another compiler mechanism with similar power. We will use the function overload resolution mechanism instead. For this to work we must take advantage of a surprising property of the `sizeof` operator. Namely, `sizeof` can tell us the size of the return type of any function at compile time. So with the help of sizeof we can subvert the function overload resolution mechanism to write metafunctions that can discriminate a subset of types.

3.1 IS_PTR

Let us begin with `IS_PTR`. `IS_PTR` must discriminate between pointer types and all others. We pass an instance of `T` named `t` to a pair of overloaded functions. It helps to think of these functions as discriminators. Each function has a uniquely sized return type so that `sizeof` can determine which function was selected by the overload resolution mechanism. So here is a simplified version of `IS_PTR` (see Appendix or [2] for a more detailed implementation).

```
/* These are the discriminating functions.
Note that only a declaration is required by
sizeof. */

char IsPtr(
  const volatile void* const  volatile);
int IsPtr(...);

// This template metafunction accepts a type T
// then sets RET to true exactly when T is a
// pointer.

template <typename T>
struct IS_PTR {
  static T t; //definition not required.
  enum { RET = (sizeof(IsPtr(t) == 1) };
};
```

3.2 IS_PTR_TO_POD

For `IS_PTR_TO_POD` we again make use of the overload function resolution mechanism. But this time we use it to "remove" the pointer before applying the `IS_POD` metafunction. The key point is that we can give the `IS_POD` metafunction access to the underlying type by using it in the return type of our discriminator. The `IF` is used as a way to convert the enum returned by `IS_POD` into a type suitable for passing to sizeof.

```
template <typename T>
struct IS_POD {
  enum {RET = /* see Appendix  */ };
};

template <typename T>
struct IS_PTR_TO_POD
{
  template <typename V>
  IF<IS_POD<V>::RET, char, int>::RET
      RemovePtr(V*);
  int RemovePtr (…);

  enum {RET =
      sizeof(RemovePtr(t) )== sizeof(char) };
  static T t;
};
```

3.3 Generalizations

It is important to note that these techniques are not limited to recovering value types from pointers. We can use the function overload mechanism to recover any type associated with a template parameter. Of course we can't use the recovered type directly. But we can hand it to a metafunction and recover an integer's worth of information from it through the sizeof operator. For example we can create such monsters as IS_VECTOR_OF_PTRS_TO_LIST_OF_PODS if we are so inclined:

```
template <typename T>
struct IS_VECTOR_OF_PTRS_TO_LIST_OF_PODS
{
  template <typename V>
  IF<IS_POD<V>::RET, char, int>::RET
      Remove(std::vector<std::list<V>*>);
  int Remove (…);

 enum {RET =
  sizeof(Remove(t))==  sizeof(char) };
  static T t;
};
```

Nor are we limited to single argument metafunctions:

```
template <typename T>
char IsSameType(T, T);
int  IsSameType(...);

template <typename T, typename U>
struct IS_SAME_TYPE {
    static T t;
    static U u;
    enum {RET =
       (sizeof(IsSameType(t, u)) == 1)};
};
template <typename T, typename U>
struct IS_PTR_TO_SAME_TYPE {
    static T t;
    static U u;

    template <typename V, typename W>
    static IF<IS_SAME_TYPE<V,W>::RET,
       char, int>::RET
       Remove(
          const volatile V* const volatile,
          const volatile W* const volatile);
    static int Remove(...);
    enum {RET =
       (sizeof(Remove(t, u)) == 1)};
};
```

3.4 Limitations

The usual template metaprogramming caveats apply: increased compile times, difficult error messages (see Appendix for one error reporting mechanism). These techniques need some refinement to properly handle reference types, see [2].

4 Related Work

The meta IF partial specialization workaround appeared in [1]. The boost type_traits library contained a version of is_ptr and many other useful metafunctions, which used to require partial specialization support from the compiler [2]. Alexandrescu brings out the power of the sizeof operator in [3]. In September 2000 Jesse Jones and I posted an earlier version of this paper including the IS_PTR sizeof technique to the boost mailing list [4]. See also opensource.adobe.org. In October 2000 John Maddock refined these results to remove the partial specialization requirement in much of the boost type_traits library, again see the thread beginning at [4]. We have used these mechanisms in a commercial product and its associated software development kit [5]. We hope that some of these techniques enable portability improvements in other libraries, for example [6].

5 Appendix. A more complete Example std::copy

Here we present a self contained implementation of an optimizing std::copy generator. This code was tested on Microsoft Visual C++ version 6 service pack 5 and version 7 beta 2.

```
#include <cstddef>
#include <cstring>

#ifdef _MSC_VER
  #pragma warning(disable: 4786)
  /* Long names truncated in browser */
#endif

namespace Meta {

// ----- META IF ----
struct FirstPicker {
 template <typename First, typename Second>
  struct Result {
      typedef First RET;
  };
};

struct SecondPicker {
  template <typename First, typename Second>
  struct Result {
      typedef Second RET;
  };
};

template <bool condition>
struct BoolToPicker {
  typedef SecondPicker RET;
};

template <>
struct BoolToPicker<true> {
  typedef FirstPicker RET;
};

template <bool condition, typename IfType,
typename ElseType>
class IF {
/* this version of if does not require partial
specialization */
  typedef typename BoolToPicker<condition>::RET
Picker;
public:
  typedef typename Picker::template
Result<IfType, ElseType>::RET RET;
};

#ifdef _MSC_VER
#pragma  warning( disable : 4660)
  template class IF<1, char, int>;
  template class IF<0, char, int>;
#endif

/* ----- META_ASSERT ----- */
struct ValidCode {};  /* type used by code that
is OK */

template <bool PREDICATE, typename ERR_MESG>
  /* metafunction that returns a type
   that cannot be instantiated if the predicate
is true */
struct ASSERT_SELECTOR
{
  typedef typename Meta::IF<PREDICATE,
ValidCode, ERR_MESG>::RET RET;
};

template <bool PREDICATE, typename ERR_MESG>
  /* causes a compile time error if the
predicate is false */
```

```
void META_ASSERT()
{
  ASSERT_SELECTOR<PREDICATE, ERR_MESG>::RET
      checker;
  (void) checker;
}

// ----- IS_PTR -----

struct IsPtrShim {
 /* allows IsPtr to detect only ptrs not
     classes with conversion operators */
 IsPtrShim(
      const volatile void* const volatile);
};


char IsPtr(IsPtrShim);
int IsPtr(...);

template <typename T>
struct IS_PTR {
  static T t;
  enum { RET = (sizeof(IsPtr(t)) == 1) };
};


// ----- IS_SAME_TYPE -----

template <typename T>
char IsSameType(T, T);
int  IsSameType(...);

template <typename T, typename U>
struct IS_SAME_TYPE {
  static T t;
  static U u;
  enum {RET = (sizeof(IsSameType(t, u)) == 1)};
};

template <typename T, typename U>
struct IS_PTR_TO_SAME_TYPE {
  static T t;
  static U u;

template <typename V, typename W>
  static
  IF<IS_SAME_TYPE<V,W>::RET, char, int>::RET
      Remove(const volatile V* const volatile,
          const volatile W* const volatile);
  static int  Remove(...);
  enum {RET = (sizeof(Remove(t, u)) == 1)};
};

// ---- IS_POD support ----
struct base_type
{ typedef  base_type   data_type;};
struct object_type
{ typedef  object_type data_type;};


template <typename T>
class PODTraits {
public:
  typedef typename Meta::IF<
      Meta::IS_PTR<T>::RET,
      base_type,
      object_type>::RET data_type;
};

// Would like to use TypeLists
// instead of macros here

#define DECLARE_BASE_TYPE(type)    \
template <>                        \
class PODTraits<type > {           \
public:                            \
  typedef base_type data_type;     \
}

//    Specializations for common base types

DECLARE_BASE_TYPE(bool);
DECLARE_BASE_TYPE(char);
DECLARE_BASE_TYPE(short);
DECLARE_BASE_TYPE(int);
DECLARE_BASE_TYPE(long);
DECLARE_BASE_TYPE(unsigned char);
DECLARE_BASE_TYPE(unsigned short);
DECLARE_BASE_TYPE(unsigned int);
DECLARE_BASE_TYPE(unsigned long);
#ifdef MACINTOSH
DECLARE_BASE_TYPE(long long);
DECLARE_BASE_TYPE(unsigned long long);
#else
DECLARE_BASE_TYPE(__int64);
DECLARE_BASE_TYPE(unsigned __int64);
#endif
DECLARE_BASE_TYPE(float);
DECLARE_BASE_TYPE(double);
DECLARE_BASE_TYPE(long double);

DECLARE_BASE_TYPE(const void*);


// ----- IS_POD -----
template <typename T>
struct IsPOD {
  enum {RET = false};
};

template <>
struct IsPOD<base_type> {
  enum {RET = true};
};

template <typename T>
struct IS_POD {
  enum {RET =
      IsPOD<PODTraits<T>::data_type>::RET};
};
template <typename T>
struct IS_PTR_TO_POD {
  template <typename V>
  static IF<IS_POD<V>::RET, char, int>::RET
      Remove(const volatile V* const volatile);
  static int  Remove(...);
  static T t;
  enum {
      RET = (sizeof(Remove(t)) == 1) };
```

```
};


} //end namespace Meta


// ---- FastCopy and SafeCopy ----
using namespace Meta;


template <typename InputIter,
  typename ForwardIter>
struct SafeCopy {
  static ForwardIter copy(
      InputIter first, InputIter last,
      ForwardIter result);
  static ForwardIter copy_backward(
      InputIter first, InputIter last,
      ForwardIter result);
  static ForwardIter uninitialized_copy(
      InputIter first, InputIter last,
      ForwardIter result);
};

template <typename InputIter,
  typename ForwardIter>
struct FastCopy {
  static ForwardIter copy(
      InputIter first, InputIter last,
      ForwardIter result);
  static ForwardIter copy_backward(
      InputIter first, InputIter last,
      ForwardIter result);
  static ForwardIter uninitialized_copy(
      InputIter first, InputIter last,
      ForwardIter result);
};


template <typename InputIter,
  typename ForwardIter>
inline ForwardIter
SafeCopy<InputIter, ForwardIter>::copy(
  InputIter first, InputIter last,
  ForwardIter result)
{
  for (; first != last; ++first, ++result)
      *result = *first;
  return result;
}

class FastCopy_on_bad_types {
  /* not instantiable (private default ctor)
*/
  FastCopy_on_bad_types() {}
};

template <typename InputIter,
  typename ForwardIter>
inline ForwardIter
FastCopy<InputIter, ForwardIter>::copy(
  InputIter first, InputIter last,
  ForwardIter result)
{
  META_ASSERT<
      IS_PTR_TO_POD<InputIter>::RET
      && IS_PTR_TO_POD<ForwardIter>::RET
      && IS_PTR_TO_SAME_TYPE<InputIter,
          ForwardIter>::RET,
      FastCopy_on_bad_types>();

  ptrdiff_t count = last - first;
  memmove(result, first, count*sizeof(*first));
  return result + count;
}


template <typename InputIter,
  typename ForwardIter>
inline ForwardIter copy(
  InputIter first, InputIter last,
  ForwardIter result)
{
  return IF<
      IS_PTR_TO_POD<InputIter>::RET
      && IS_PTR_TO_POD<ForwardIter>::RET
      && IS_PTR_TO_SAME_TYPE<InputIter,
          ForwardIter>::RET,
      FastCopy<InputIter, ForwardIter>,
      SafeCopy<InputIter, ForwardIter>
  >::RET::copy(first, last, result);
 }
```

6 References

[1] K. Czarnecki and U. W. Eisenecker. Generative Programming. Methods, Tools, and Applications. Addison-Wesley 2000

[2] boost/type_traits http://www.boost.org/libs/type_traits/index.htm

[3] Alexandrescu sizeof C Users Journal experts 10/00 http://www.cuj.com/experts/1810/alexandr.htm?topic=experts

[4] original internet post September 2000 http://groups.yahoo.com/group/boost/message/5441 http://opensource.adobe.com

[5] K2Vector in the Adobe Indesign 2.0 SDK http://www.adobe.com/products/indesign

[6] STLPort possible use of type_traits 07/01 http://www.stlport.com/dcforum/DCForumID10/5.html#3