



Simulating Partial Specialization

Mat Marcus

mmarcus@adobe.com

October 15, 2001



Overview

- **The focus is on solving problems using generative template metaprogramming in a shrink wrap software development environment.**
- **Will work through two sample problems that arise when implementing generic containers in C++**
- **Along the way will sketch two results from our research. One is a year old and one is a week old, so it is still somewhat rough.**



Context

- **Shrink wrap Software Development**
- **Deliver for Mac OS and Windows**
- **Associated SDK**
 - Software Is Developed in C++
 - Extensibility is critical
 - API mostly uses interface classes
 - Binary compatibility across releases is important
- **Microsoft compiler has limited support for standard C++**
 - In particular no support for partial specialization



Motivation

- **Would like to use generic containers from Standard Template Library**
 - Powerful, convenient, well documented framework
- **Couldn't use vendors implementations**
 - Existing STL implementations suffered from template code bloat under MSVC - no partial specialization
 - Binary compatibility across compiler versions would suffer
- **Existing Internal Container Libraries**
 - Optimized, limited template bloat
 - Non standard, not easy to use
 - Too many classes to choose from



Problem

- **Implementing part of a `std::vector` clone**
- **Two of the non-functional requirements**
 - Require no template code bloat
 - Require optimized `std::copy`, etc. for “plain old data” types, e.g. built-in types



Naïve Approach Causes Code Bloat

```
/* Bloat: The compiler will instantiate separate code
for each vector of pointers even though the machine
instructions are the same. */
```

```
template <typename T>
class SimpleVector
{ /* . . . */};
```

```
/* This is often addressed by creating a separate
container template */
```

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

```
/* But now clients must remember to select the
appropriate container template, PtrVector, if they are
using pointers. */
```



A Solution: Use a Vector Generator

- **We will define a container generating metafunction:**

- ```
template <typename T>
struct VECTOR_GENERATOR {
 typedef /* . . . */ RET;};
```

- **VECTOR\_GENERATOR<T> :: RET will be the appropriate container**

- VECTOR\_GENERATOR<int\*> :: RET is of type PtrVector<int\*>
  - VECTOR\_GENERATOR<std::string> :: RET is of type SimpleVector<std::string>

- **Can add some syntactic sugar using inheritance**

- ```
template <typename T>
class Vector : public typename
VECTOR_GENERATOR<T>::RET
{ /* implement delegating ctors */ }
```
 - Now client can write the code as if partial specialization was present without fear of template bloat e.g: Vector<int*> v;



Vector Generator Implementation

- **Create metafunction IS_PTR**

- E.g. IS_PTR<int *> :: RET == 1

- **Borrow template metafunction IF [CE2000]**

- E.g. IF<1+1 == 3, int, std::string> :: RET is of type std::string

- **Then VECTOR_GENERATOR can be written:**

- ```
template <typename T>
struct VECTOR_GENERATOR {
 typedef typename IF<IS_PTR<T>::RET,
 PtrVector<T>,
 SimpleVector<T>
 >::RET RET;
};
```



## IS\_PTR

- **There were existing versions of IS\_PTR but all relied on partial template pointer specialization.**
- **We can subvert the function overload resolution mechanism instead.**
- **Alexandrescu pointed out the usefulness of `sizeof` in template programming. In particular `sizeof` can determine the size of any expression at compile time.**



## IS\_PTR implementation

```
/*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(void*);
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)) == sizeof(char)) };
};
```



## “Associated” Type Recovery

- We would also like to recover the underlying value type. That is we would like to be able to write something like `REMOVE_PTR<T>::RET` otherwise known as `std::iterator_traits<T>::value_type`
- We know of no way to do so for pointers without partial specialization. But we don't always need to completely recover the underlying type. Sometimes it suffices to “ask it a question”.
- In those cases we can turn our client code “inside out”. If we have some work to carry out we can create a metafunction to be executed with the underlying type as its argument. The restriction is that the metafunction can only return an enum not a type.



## Applying a metafunction to an Associated Type

- As an example suppose we have the metafunction `IS_POD` which detects whether a type `T` is “plain old data”. Lets say we want to detect whether `T` points to plain old data. That is we would like to have a metafunction `IS_PTR_TO_POD`. If we had `REMOVE_PTR` we might be able to use `IS_POD<typename REMOVE_PTR<T>::RET>::RET`. Instead we can ask `IS_POD` to be evaluated in the context of the underlying type.



## Applying a metafunction to an Associated Type (cont.)

- Note that we are not limited to recovering value types from pointers. We can apply the same technique to arbitrary type relationships. Nor are we limited to single argument metafunctions. For example we can ask:
  - `IS_VECTOR_OF_LIST_OF_POD<T>`
  - `IS_PTR_TO_SAME<T,U>`



## Applied Inside Out Type Recovery

- As an example once we have our `IS_PTR_TO_POD` we can implement our other non-functional requirement - an optimized `std::copy`.



```
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);
}
```

Generative Programming Workshop  
OOPSLA 2001



## IS\_PTR\_TO\_POD implementation

/\* For IS\_PTR\_TO\_POD we again make use of the function overload 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. \*/

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

template <typename T>
struct IS_PTR_TO_POD
{
 /* The IF below is one way to convert the enum returned
 by IS_POD into a type suitable for passing to sizeof. */

 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;
};
```

Generative Programming Workshop  
OOPSLA 2001



## Related Work

- The IF for broken compilers appeared in [CE2000].
- IS\_PTR existed for some time for compilers that support partial specialization, see boost::type\_traits for example.
- In September 2000 Jesse Jones and I published an implementation for “broken” compilers using properties of the sizeof operator pointed out by Alexandrescu.
- These “workarounds” were refined by John Maddock into boost::type\_traits for is\_pointer and similar metafunctions.
- Last week when preparing this paper I uncovered the “inside out” type recovery technique.



## Discussion ideas

- Why aren’t explicit control constructs (IF, SWITCH, WHILE, etc.) used more widely by the template metaprogramming community. This style from [CE2000] seems more readable than template specialization. In addition it can guide us in writing metaprograms that would otherwise seem too complex or escape our notice.
- We can avoid some uses of std::iterator\_traits<T>::value\_type by turning calculations inside out to “recover the associated type”. Where else might it be this technique be useful.



**Q&A**



**Adobe**

**everywhere**  
you look™

