INTRODUCTION INTO C++ TEMPLATE META PROGRAMMING

Motivating Example

Create a string for logging of content of any value …

```
template <typename T>
std::string printValue(T const& v);
```

- Any Type may has an operator <<
- Any Type may be derived from Formattable
- Any Type may have an implementation of PrintTo function

Let's specialize

```
template <typename T>
std::string printValue(T const& v);

template <>
std::string printValue(std::string const& v) {
  return v;
}

template <>
std::string printValue(MString const& v) {
  return v.toStdString();
}
```

Implement for all types …

This is rather boring!

Error prone!

New types need special care!

Next try – let's delegate the work

```
template <typename T>
struct Printer;

template <typename T>
std::string printValue(const T& v){
  std::ostringstream os;
  Printer<T>::print(v, os);
  return os.str();
}
```

Let’s do it for std::string

```
template <>
struct Printer<std::string>
{
  static void print(std::string const& v,
                    std::ostream& os) {
    os << v;
  }
};
```

Nothing gained so far…

But let's change the `struct Printer` a bit:

```
template <typename T, typename Enabled = void>
struct Printer;
```

What is “typename Enabled = void” ?

```
template <typename T,
          typename Enabled = void>
struct Printer;
```

- Is only taken into account during specialization if the type “Enabled” is a real type
- So use it as a switch within a switching construct
  - C++11’s switch `std::enable_if<U>::type`
  - boost’s switch `boost::enable_if<U>::type`
- U must be `true-` or `false-type`

Let‘s make a switch

```
template<bool B, class T = void>
struct enable_if {};
```

Partially specialization for true:

```
template<class T>
struct enable_if<true, T>
{
  typedef T type;
};
```

What‘s a true or false type?

```
struct true_type
{
  static const bool value = true;
};

struct false_type
{
  static const bool value = false;
};
```

Let’s use a partial specialization for all types that have a string stream operator

```
template <typename T>
struct Printer<T,
               typename std::enable_if<
                 has_string_stream_operator<T>::value,
                 T>::type
               >
{
  static void print(T const& v, std::ostream& os) {
    os << v;
  }
};
```

Who needs PrintTo()?

- Preferred interface in GoogleTest to print values
- Used by GoogleTest to print values in case an assertion failed

```
void PrintTo(int v, std::ostream* s);
```

More specialization …

```
template <typename T>
struct Printer<T,
               typename std::enable_if<
                 has_PrintTo<T>::value,
                 T
               >::type
              > {
  static void print(T const& v, std::ostream& os){
    PrintTo(v, &os);
  }
};
```

Classic SFINAE

(Specialization failure is not

```
template <typename T>
class has_PrintTo
{
  typedef char   Yes;
  typedef struct No { char dummy[2]; };

  template<typename U>
  static Yes test(U* p);

  template<typename>
  static No test(...);
public:
  static const bool value =
    sizeof(test<T>(nullptr)) == sizeof(Yes);
};
```

```
typedef char Yes;
typedef int  No;
```

This may not work, because standard does not require sizeof(int) != sizeof(char)

The general form is not enough!

Get a little help from decltype

```
template <typename T>
class has_PrintTo
{
  typedef char   Yes;
  typedef struct No { char dummy[2]; };
  static std::ostream os;

  template<typename U>
  static decltype(PrintTo(*p, &os), Yes(0)) test(U* p);

  template<typename>
  static No test(...);

public:
  static const bool value =
    sizeof(test<T>(nullptr)) == sizeof(Yes);
};
```

Comma Operator

Unfortunately does not compile!

Final has_PrintTo

```
template <typename T>
class has_PrintTo
{
  typedef char   Yes;
  typedef struct No { char dummy[2]; };
  static std::ostream os;

  template<typename U>
  static auto test(U* p) -> decltype(PrintTo(*p, &os), Yes(0));

  template<typename>
  static No test(...);

public:
  static const bool
    value = sizeof(test<T>(nullptr)) == sizeof(Yes);
};
```

has_string_stream_operator

```
 template <typename T>
class has_string_stream_operator {
  typedef char   Yes;
  typedef struct No { char dummy[2]; };
  static std::ostream os;

  template<typename U>
  static auto test(U* p) -> decltype(os << *p, Yes(0));

  template<typename>
  static No test(...);

public:
  static const bool
    value = sizeof(test<T>(nullptr)) == sizeof(Yes);
};
```

With a little help of std::is_base_of for Formattable types

```
template <typename T>
struct Printer<T,
               typename std::enable_if<
                 std::is_base_of<Formattable,
                 T>::value
              >::type> {
    static void print(T const& v, std::ostream& os){
      Formatter f;
      v.formatTo(f);
      os << f.getString().toStdString();
    }
  };
```

printValue now can take any value of a type that

- has a string stream operator<<
- or has a PrintTo(T const&, std::ostream*) function
- or is Formattable

Let’s write a UnitTest!

Test

- PrintTo
- stream operator
- and for other traits

Done!

Used it in real code…

Does not compile for std::string

???????

But std::string has operator<<!!!

std::string is a typedef :

```
typedef basic_string<
  char,
  char_traits<char>,
  allocator<char>> string;
```

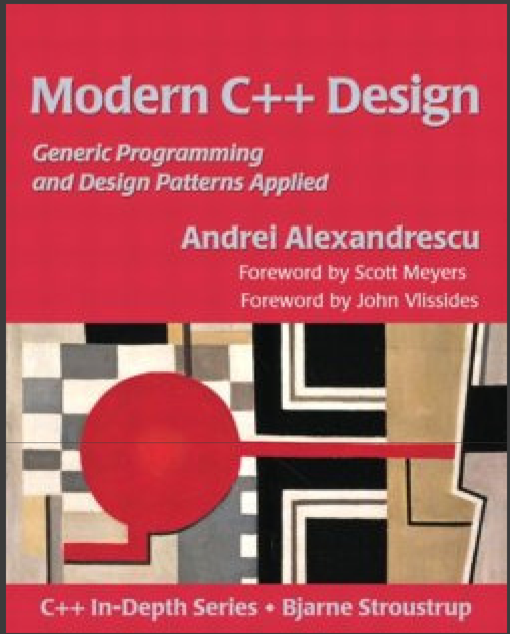
So we still need the specialization for std::string

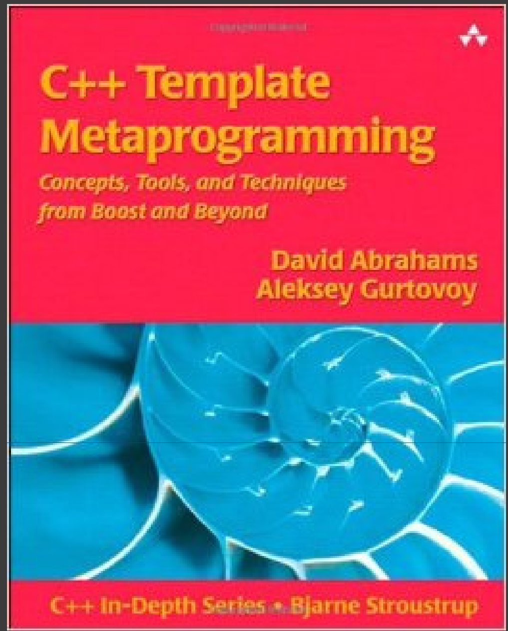
```
template <>
struct Printer<std::string>
{
  static void print(std::string const& v,
                    std::ostream& os) {
    os << v;
  }
};
```

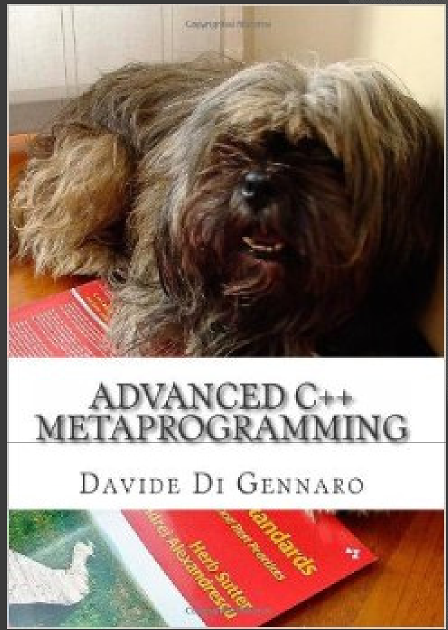
Examples

```
log(printValue(1));
log(printValue(string(“Foo“));
log(printValue(Bar(42,4711));
```

Reference







Introduction into modern C++ Techniques

Presentation by Michael Caisse at C++ Now 2012

Introduction into auto and declspec

Blog by Thomas Becker

Great source of knowledge

www.stackoverflow.com

Thank’s for your attention

Feedback is as always welcome!

felix@petriconi.net