Meta-Programming with C++
More Meta-Programming with C++11

Thomas Gschwind
<thg@....ibm....>
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Meta-Programming with Templates
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Introduction

- Meta Programming Definitions
  - Programming that manipulates program entities
  - Programs that compute at compile time and generate programs

- Why would we want to do this?
  - Improved type safety
  - Improve runtime performance by computing values at compile-time
  - Improve code readability
Generic Programming vs Meta Programming

- Theoretically,
  - any template instantiation is meta-programming
  - any use of the pre-processor is meta-programming
  - all we do is write generic programs

- Practically, the intent indicates whether it is meta-programming

- When implementing a generic type (min, find_if)
  - I would not feel like writing a meta-program
  - The intent of the exercise is to write one algorithm that is the same for many different types

- When writing a meta-program (difference)
  - I want that the compiler takes a decision at compile time
  - Whether to use the forward iterator or random access iterator algorithm
Agenda

- Introduction

- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Meta-Programming with Templates

- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Binders - Motivation

- Want to find an element in a container that fulfills a criterion
- For instance, first element in the container that is less than a given value
- In Java
  - If the container provides this fine (unlikely)
  - If not, we start implementing our own routine
Locating an Element

```cpp
int less_than_17(int x) {
    return x<17;
}

void foo(vector<int> v) {
    vector<int>::iterator b=r.begin(), e=r.end(), i;
    i=find_if(b, e, less_than_17);
    ...
}
```

- Having to write that helper function is bothersome
- How can we improve this (without using C++11 lambdas)
Function Objects

Predicates <functional>
- equal_to, not_equal_to
- greater, greater_equal, less, less_equal
- logical_and, logical_or
- logical_not (unary)

Arithmetic Operations <functional>
- plus, minus, multiplies, divides, modulus
- negate (unary)
Locating an Element

```cpp
void foo(vector<int> v) {
    vector<int>::iterator b=r.begin(), e=r.end(), i;
    i=find_if(b, e, bind2nd(less<int>(),17));
    ...
}
```

- We say we want that the greater than function is executed
- Problem greater than takes two arguments
- Solution, the bind2nd function binds one argument to a given value
bind2nd(binop, arg2)

- Binds the second argument of a function
- If I have already a function less/2 one would not like to write another one less/1 for all possible arguments

- Implementation
  - Need to store a binary operation binop and the second argument arg2
  => We need a function object to implement this
our_binder2nd

template <typename BinOp>
class our_binder2nd {
  protected:
    BinOp op;
    Arg2 arg2;
  public:
    our_binder2nd(BinOp o, Arg2 a2) : op(o), arg2(a2) {}
    Res operator() (Arg1 arg1) { return op(arg1, arg2); }
};

Solution

- Function Object Bases
- They provide standardized names for arguments, and return types
Function Object Bases

- Provide standardized names for arguments, and return types for function objects
- Use them religiously!

```cpp
template <class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};
template <class Arg, class Res> struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg second_argument_type;
    typedef Res result_type;
};
```
bind2nd

template <class BinOp>
class binder2nd : public
    unary_function<BinOp::first_argument_type, BinOp::result_type>
{
  protected:
    BinOp op;
    typename BinOp::second_argument_type arg2;
  public:
    binder2nd(const BinOp &o,
               const typename BinOp::second_argument_type &a2)
        : op(o), arg2(a2) {}
    result_type operator()(const argument_type &arg1) {
        return op(arg1, arg2); }
};

template<class BinOp, class T> binder2nd<BinOp>
bind2nd(const BinOp &o, const T &v) {
    return binder2nd<BinOp>(o, v); }
`find_if(\ldots, bind\_2nd(less, 17))`

```cpp
find_if(I f, I l, Op op)  
{ while(f!=l) op(*f++); }
```

```cpp
op = binder\_2nd{op=less, a=17}
```

```cpp
op\_operator\_() (T x)  
{ return op(x, a); }
```

```cpp
less(x, 17)
```
Binders, Adapters, Negaters

**Binders** <functional>
- `bind1st`, `bind2nd`

**Adapters** <functional>
- `mem_fun`, `mem_fun_ref`, `ptr_fun`

**Negaters** <functional>
- `not1`, `not2`
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Template Meta-Programming
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Motivation

- Callback functions can be expensive and difficult to read
  - \( a = \text{integrate} \text{\texttt{evaluatef}}, \text{\texttt{0.0, 10.0}}; \)

- Multiplication of vectors and arrays is either very expensive or very unreadable
  - Fusion of loops
  - Vector \( a(\ldots), b(\ldots), c(\ldots); c = a*b+c; \)
  - Matrix \( a(\ldots), b(\ldots), c(\ldots); c = a*(b+c); \)

- Use templates as a mechanism to change how the program is being compiled
  - Also known as meta-programming
  - Yes, templates are not necessarily the most readable mechanism for this
Function Objects

- Inline expansion possible
- Not always possible
- Sometimes *unreadable*

```cpp
template <class Op>
double integrate(Op op, double x0, double x1) { ... }

double evaluatef(double x) { return 2.0*x/(1.0+x); }

void foo() {
    ...  
    cout << integrate(evaluatef(), 0.0, 10.0) << endl;
}
```
Expression Templates

- Allow expressions to be passed as function argument
  - \( a = \text{integrate}(2.0 \times x/(1.0+x), \ 0.0, \ 10.0) \)

- This idea is nothing new, binders work similar
  - \( i = \text{find_if}(..., \text{bind\_2nd}(\text{less}(), 17)) \)

- Taking the expression apart
  - \( 2.0 \times x \) would be similar to \( \text{bind\_1st}(\text{mul}, 2.0) \)
  - \( 1.0 + x \) would be similar to \( \text{bind\_1st}(\text{add}, 1.0) \)

- How do we model the division?
template <class Op, class O1, class O2>
struct combineops_t :
public unary_function<O1::arg_t, Op::res_t> {
    Op op; O1 o1; O2 o2;
    combineops_t(combineops_t(Op binop, O1 op1, O2 op2)
        : op(binop), o1(op1), o2(op2) {"
    res_t operator()(arg_t x) {
        return op(o1(x), o2(x));
    }
};

template <class Op, class O1, class O2>
struct combineops_t
combineops(0p op, O1 o1, Op2 o2) {
    return combineops_t<Op,O1,O2>(op,o1,o2);
}
combineops: Usage

```plaintext
a=integrate(combineops(div,
    bind_1st(mul,2.0),
    bind_1st(add,1.0)),
    0.0, 10.0);
```

- This expression is not yet more readable BUT soon it will be
**combineops: Readability I**

New (unary) function objects

- `literal_t/literal` returning a constant
- `identity_t/identity` returning \( x \) (the argument itself)

```cpp
void foo() {
    identity_t<double> x();
    a=integrate(combineops(div,
        combineops(mul,literal(2.0),x),
        combineops(add,literal(1.0),x)),
            0.0, 10.0);
    cout << a << endl;
}
```
Expression Tree

```
+  literal  identity
  /  combineops  op  o1  o2
  *  literal  identity

combineops  op  o1  o2
```
**combineops: Readability II**

- Define operators /, *, ... returning the according `combineops` objects

```cpp
template<class T>
combineops_t<mul, literal_t<T>, identity_t<T> >
operator* (literal_t<T> l, identity_t<T> i) {
    typedef combineops_t<mul, literal_t<T>, identity_t<T> > r;
    return r(mul,l,i);
}

// define *, /, ... for
// literal_t and combineops_t
// identity_t and identity_t
// identity_t and combineops_t
// ...
```
combineops: Usage

```cpp
void foo() {
    identity_t<double> x();
    double a;
    a=integrate(literal(2.0)*x/(literal(1.0)+x)), 0.0, 10.0);
    cout << a << endl;
}
```

- Looks good?
- `literal(2.0)` could be written as 2.0 if `operator*(double, ...)` were defined as well
Summary

+ Achieved our goal
- Somewhat clumsy to define all the different operator combination
- Error prone
Second Approach

- First approach error prone
  - Too many combinations of argument types for +, /, … operators
Expression Tree (1st)

```
combineops op o1 o2
\/
combineops op o1 o2
*
literal identity
+
literal identity
```

IBM Research – Zurich
Business Integration Technologies

© 2010 IBM Corporation
Expression Tree (2nd)
ET for Vectors

- Vector a(...), b(...), c(...); c=a*b+c;
- Similar to expressions but using iterators
- So where do we place the loop to iterate over all the elements?

```cpp
template<class A>
Dvec& Dvec::operator=(DVExpr<A> expr) {
    for(iterator i=begin(), last=end(),
        i!=last;
        i++, expr++) *i=expr(); // i=*expr in Veldhuizen
    return *this;
}
```
ET for Matrices

- Matrix a(...), b(...), c(...); c = a*b + c;
- See “Generic Programming in POOMA and PETE”…”
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Meta-Programming with Templates
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Templates with Non-Typenames

- Classes can not only be parameterized over a specific type but also over a specific int (or whatever)

```cpp
template<
typename T, int SZ>
class array {
    T[SZ] rep;

public:
    T &operator[](int i) {
        if(i>SZ) { /* handle error */ }
        else return rep[i];
    }

    ...
}
```
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Template Meta-Programming
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Meta-Programming with Templates

- What does this code do? Where would it be useful?
Meta-Programming with Templates

- Yes, the code checks whether the number is a prime number

```cpp
template<int P, int N> struct isprime2 {
    static const int res=isprime2<P,P%N?N-1:0>::res;
};

template<int P> struct isprime2<P,1> {
    static const int res=1; }

template<int P> struct isprime2<P,0> {
    static const int res=0; }

template<int N> struct isprime {
    static const int res=isprime2<N,N-1>::res; }

int main() {
    cout << isprime<3>::res << "," << isprime<4>::res << "," << isprime<5>::res << endl; }
```
Meta-Programming with Templates

- Where is the previous code useful?
- If we need somewhere a prime if we add a template to compute the next prime

```cpp
template<typename T> class my_hash_table {
  T table[compute_next_prime<20000>::res];
  ...
};
```
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Template Meta-Programming
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Meta-Programming in C++11

- Expression Templates as in the integration sample
  - Largely not necessary since we have lambda functions

- If we want to do things like vector multiplications
  - Expression template syntax is still nicer to use
  - Logic hidden in the assignment operator
  - No lambda expression just Vector v=a*b;

- For matrices, we cannot use lambda expressions
  - Loops need to be enrolled in an interleaved form
Meta-Programming in C++11

- C++11 makes our life easier (and more complicated again)
- Meta-programming with constexpr

```cpp
constexpr bool isprime2(int i, int n) {
    return (n%i==0) ? false
        : (i*i<n) ? isprime2(i+2,n) : true;
}
constexpr bool isprime(int n) {
    return (n%2==0) ? (n==2) : isprime2(3,n);
}
constexpr int nextprime(int i) {
    return isprime(i) ? i : nextprime(i+1);
}
int main(int argc, char *argv[]) {
    constexpr int res=nextprime(1234567890);
    cout << res << endl;
}
```
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Template Meta-Programming
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Concepts

- In C++, we can encode concepts
- These concepts can be used to limit the scope of our templates
  - `is_copy_assignable<T>::value`: true if type T may be copied (readily supported in C++11)
  - `(LessThanComparable<T>)` compilation will only succeed if T is `LessThanComparable` (from boost.org)

```cpp
#include <type_traits>
#include <boost/concept_check.hpp>

template<typename T>
class rpn_calculator {
    static_assert(is_copy_assignable<T>::value,
                  "rpn_calculator's number type must be a copy assignable type");
    BOOST_CONCEPT_ASSERT((boost::LessThanComparable<T>));
    ...
```
Concepts (cont’d)

- Concepts allow us to catch errors early
- Somewhere we instantiate our rpn_calculator
  - The rpn_calculator is parameterized with complex<float>
  - Somewhere the rpn_calculator uses a min function (again a template)
  - Somewhere the min function uses < (less_than) to identify the minimum
  - Complex numbers do not support < (less_than) which is an error
- Without concepts, the C++ compiler will tell us all this
  - (194 lines of errors in my rpn_calculator implementation)
  - Need to sift through the “entire” implementation to understand the error
  - Now, imagine there may have been more indirections
- Concepts make the compiler fail early
  - At the static_assert
  - The implementor knows what is necessary, typically we do not care why
Concepts (cont’d)

- We can implement more readable/useful concepts as follows
  - `CopyAssignable<T>()`: true if type T may be copied
  - `Ordered<T>()`: true if T has an ordering

- There “should” be already libraries available that do this (Bjarne Stroustrup. The C++ Programming Language, 4th Ed.)

```cpp
#include <type_traits>

template<typename T>
class rpn_calculator {
    static_assert(CopyAssignable<T>(),
                  "rpn_calculator's number type must be a copy assignable type");
    static_assert(Ordered<T>(),
                  "rpn_calculator's number type must be an ordered type");
    ...
```
Implementation of CopyAssignable<T>() and Ordered<T>()

```cpp
#include <type_traits>

template<typename T> constexpr int CopyAssignable() {
    return is_copyAssignable<T>::value;
}

template<typename T> struct is_ordered {
    enum { value = 0 };
};

template<> struct is_ordered<int> {
    enum { value = 1 };
};

template<> struct is_ordered<long> {
    enum { value = 1 };
};

template<> struct is_ordered<float> {
    enum { value = 1 };
};

template<> struct is_ordered<double> {
    enum { value = 1 };
};

template<typename T> constexpr int Ordered() {
    return is_ordered<T>::value;
}
```

These definitions should be provided as part of a library.
Optimizing rpn_calculator

- Concept checking makes life easy
- In case of the complex numbers not being able to use the rpn_calculator at all may not be very rewarding
- Ideally, we want to disable the minimum function
rpn_calculator with optional min (Naïve approach)

template<typename T>
class rpn_calculator {
  ...
  void run(void) {
    for(;;) {
      ...
      if (...) {
        ...
      } else if (cmd=="m" && n>=2 && Ordered<T>() { 
        T b=pop_back(), a=pop_back();
        push_back(min(a,b));
      } else {
        cerr << "Unknown command" << endl;
      }
    }
  }
}
Wrapping min

- We can create mymin as template and use partial specialization
  - Needs to be done for all non-ordered types
- C++11 provides an enable_if function that allows to selectively define functions based on a condition evaluated during compile time

```cpp
template<bool B, typename T=void>
using Enable_if = typename std::enable_if<B,T>::type;

template<typename T>  // standard wrapper for ordered types
Enable_if<Ordered<T>(), T> mymin(T x, T y) {
    return std::min(x, y); }

template<typename T>  // dummy implementation for others
Enable_if<!Ordered<T>(), T> mymin(T x, T y) { return 0; }
```
Agenda

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Template Meta-Programming
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Variadic Templates

- Wouldn’t it be nice to have printf that supports user-defined types?
- Let’s implement a simplified printf function
  - Arguments just specified as % (not %d, %g, etc)
  - % represented as %%
  - Our printf identifies the type automatically
  - Supports user-defined types
Variadic Templates (cont’d)

```cpp
void printf(const char *s) {
    if (s == nullptr) return;
    while (*s) {
        if (*s == '%' && ++s != '%') throw error("missing argument");
        cout << *s++;
    }
}

template<typename T, typename..., Args>
void printf(const char *s, T value, Args... Args) {
    if (s == nullptr) throw error("too many arguments");
    while (*s) {
        if (*s == '%' && ++s != '%') {
            cout << value; return printf(s, args...);
        }
        cout << *s++;
    }
    throw error("too many arguments"); }
```
Summary

- Introduction
- C++ Meta-Programming
  - Binders or Meta-Programming light
  - Expression Templates
  - Templates with Non-Typenames
  - Template Meta-Programming
- C++11 Meta-Programming
  - Meta-Programming with constexpr
  - Concepts
  - Meta-Programming with Concepts
  - Variadic Templates
Exercise 1

- Implement a function findIf(Iterator iter, Matcher matcher) in Java that finds the first element in the sequence defined by iter Matcher is an interface with a single method match return true if the element matches
- Implement in a separate Java file a benchmark that executes the above on a Vector with several millions of elements
- Implement the same benchmark in C++ with the C++ find_if method and, e.g., lambdas
Exercise 2

- Extend the RPN calculator such that the min function is available for ordered types but unavailable for unordered types (i.e., complex numbers).
Exercise 3

- Implement a function that merges the elements of two containers
  - Think of how to represent the containers
  - How shall the elements be added to the target containers
  - How can fundamentally different containers be merged? map and vector
  - Make use of templates
Exercise 4

- Implement an iterator that encapsulates another iterator (i.e., a sequence) and that performs range checking
- The iterator is initialized with the current element, and the first and last element of the sequence
- If the iterator points to the first element and is decreased OR if the iterator points to the last element and is increased signal an error – choose an appropriate form of signaling the error
Next Lecture

- More C++11 Features, Factories, Multi Methods, Repetitorium

Have fun solving the examples!

See you next week!