Advanced Software Engineering with C++ Templates

More Metaprogramming

Thomas Gschwind <thg at zurich dot ibm dot com>
Agenda

- Metaprogramming
  - Type Support
  - Concepts
- Expression Templates
Type Support

- Type support allows us to query types for their characteristics
  - is_array, is_enum, is_function, is_base_of, ...
  - is_standard_kayout, is_pod, ...
  - is_copy_assignable, ...

- If we have certain requirements for our template parameters, we can check this as part of our template
Type Support

- Most (all?) our RPN calculator at some point in time create a copy of the type it is calculating with
  - Hence, the type must be copy_assignable (is_copy_assignable)
  - Depending on the implementation possibly also copy_constructible (is...)

- One use of this is to generate easy to understand error messages
  - Typically, type errors, missing functions can only be identified at the point the corresponding artifact is used
  - With templates this may depend on a template instantiation
  - Which in turn may again depend on a template instantiation
  - And so on and so on...
The static assertion in combination with the error messages gives the developer an easier to understand message of what is going on.

We also get the extended error when that code is instantiated but that we can safely ignore and recompile.
Concepts

- Concepts are a further extension to the type support functions
- Express not only compiler related type attributes but concepts such as is_ordered, etc.
  - This allows us also to specify is_ordered as a requirement for our rpn_calculator as the min function depends on it.
- However, this idea is already around since C++0x (which became C++11)
  - They did not make it into C++11, or C++14, and not C++17 either
  - To play with concepts we have to revert to the BOOST Concept Library
  - Good news this library even works with good old C++03 code 😊

```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>));
...
```
Using Type Support / Concepts for Metaprogramming

- We want to implement a function different depending on its underlying type/characteristics
  - Some data types can be copied with `memcpy`
  - Other data types need to be copied by explicitly copying them (operator=)
C++11: using, the new typedef

- The typedef is nice but does not work with templates
- Using can be used to create parametric type aliases
- The below allows to write `IteratorCategoryOf<T> instead of `typename std::iterator_traits<I>::iterator_category`

```cpp
template<typename I> using DifferenceTypeOf = typename std::iterator_traits<I>::difference_type;

template<typename I> using IteratorCategoryOf = typename std::iterator_traits<I>::iterator_category;

template<typename I> using IsRandomAccessIterator = std::is_base_of<std::random_access_iterator_tag, IteratorCategoryOf<I>>;
```
Copying Memory Efficiently

- The `std::enable_if<B, T>` function evaluates to a type with `type=T` if `B==true`, otherwise to an empty type.
- An empty type leads to an invalid template substitution and is ignored according to the “Substitution Failure Is Not An Error” rule (SFINAE).

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

template<typename T>
EnableIf<std::is_pod<T>::value>::type
copy(const T *src, T *dst, unsigned len) {
    memcpy(dst, src, len*sizeof(T));
}

template<typename T>
EnableIf<!std::is_pod<T>::value>::type
copy(const T *src, T *dst, unsigned len) {
    for (unsigned i=0; i<len; ++i) *dst++=*src++;
}
```
Agenda

- Metaprogramming
  - Type Support
  - Concepts
- Expression Templates
Expression Templates: Motivation

- Callback functions can be expensive and difficult to read
  - `a=integrate(evaluatef, 0.0, 10.0);`
  - Yes, C++11 lambdas can be used to achieve the same more easily

- Multiplication of vectors and arrays is either very expensive or very unreadable
  - Fusion of loops
    - Vector `a(...), b(...), c(...); c=a*b+c;`
    - Matrix `a(...), b(...), c(...); c=a*(b+c);`
  - No, C++11 lambdas cannot be used to achieve this

- Use templates to change how the program is being compiled
  - Also known as meta-programming
  - Yes, templates are not necessarily the most readable mechanism for this
Expression Templates: Functors

- 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: Building Blocks

- 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
  
  - \( i = \text{find_if}(..., \text{bind}_\text{2nd}(\text{less}(), 17)) \)

- Taking the expression apart
  
  - \( 2.0 \times x \text{ would be similar to } \text{bind}_\text{1st}(\text{mul}, 2.0) \)
  
  - \( 1.0 + x \text{ would be similar to } \text{bind}_\text{1st}(\text{add}, 1.0) \)

- How do we model the division?
Expression Templates: combineops

```cpp
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(Op op, O1 o1, Op2 o2) {
    return combineops_t<Op, O1, O2>(op, o1, o2);
}
```
Expression Templates: Using `combineops`

```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
Expression Templates: Readability I

- Before we make it more readable, let's make it first “less” readable
- New (unary) function objects
  - `literal_t` returning a constant
  - `identity_t` 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 Templates: Expression Tree

```
/combineops op o1 o2
    *
literal identity

+combineops op o1 o2
    literal identity
```
Define operators /, *, ... returning the according `combineops` objects

```cpp
#include <iostream>

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
// ...
```
Expression Templates: 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 be written as `2.0` if `operator*(double, ...)` were defined as well
Expression Templates: Summary

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

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

combineops op o1 o2

/ combineops op o1 o2 combineops op o1 o2

* literal identity + literal identity
Expression Templates: Expression Tree (2nd Approach)

Every Problem can be solved with an extra indirection
• Encapsulate the different expressions in expr
• Only define operators for expr
Expression Templates for Vectors

- \( \text{Vector } a(...), b(...), c(...); c = a*b + c; \)
- 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();
    }
    return *this;
}
```
Expression Templates for Matrices

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

Summary

- Metaprogramming
  - Type Support
  - Concepts
- Expression Templates
Reimplement our distance function that used dynamic algorithm selection with the enable_if feature we have seen in this lecture

- Hint: If you use the using directives from slide 8 life gets easier and the code more readable
Exercise 2 (Optional)

- Words of Caution
  - The BOOST concept classes and enable_if don’t work well together which is why this exercise is optional
  - Only if you like tinkering around and you are curious, this is for you

- Extend the RPN calculator such that the min function is available for ordered types but unavailable for unordered types (i.e., complex numbers).
Schöne Weihnachten und einen Guten Rutsch ins Neue Jahr

Happy Season Greetings