Wigner GPU Lab

Introduction to Template Meta-Programming

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

D. Berényi – M. F. Nagy-Egri

The C++ Programming Language

... C++ ...

C++ is a complex language...

Languages inside C++

C++ has 4 inner languages (as far as we know):

- The „normal”, „traditional” value-level
- The type- or template-level
- The C preprocessor

- And another obscure level based on whether a function is defined already or not

Languages inside C++

C++ has 4 inner languages (as far as we know):

- The „normal”, „traditional” value-level
- The type- or template-level
- The C preprocessor
- And another obscure level based on whether a function is defined already or not

This is what we talked about up to now

This is by accident a functional language

This can be seen again as a generic language

This is... you better not want to know☺

Languages inside C++

C++ has 4 inner languages (as far as we know):

- The „normal”, „traditional” value-level
- The type- or template-level
- The C preprocessor
- And another obscure level based on whether a function is defined already or not

Lets see this

C++ templates

What are templates exactly?

C++ templates

Templates are „code templates” for the compiler.

They have „placeholders” that can be filled in later.

This „fill of placeholders” is called *instantiation*.

After instantiation, it is processed (typechecked, compiled etc.) as if it was written by hand by the programmer.

C++ templates

Example:

This is a class template definition (despite it is a struct, never mind)

```
template<typename T>
struct Circle
{
	T r;
	T area() const { return r*r*3.14; }
};
```

C++ templates

Example:
This is a class template definition (despite it is a struct, never mind)

```
template<typename T>
struct Circle
{
	T r;
	T area() const { return r*r*3.14; }
};
```

These are two class template instantiations:

```
Circle<float> c1;
Circle<double> c2;
```

C++ templates

Example: This is a class template definition (despite it is a struct, never mind)

```
template<typename T>
struct Circle
{
   T r;
   T area() const
   { return r*r*3.14; }
};
```

```
struct Circle
{
   float r;
   float area() const
   {
      return r*r*3.14;
   }
};
```

```
struct Circle
{
   double r;
   double area() const
   {
      return r*r*3.14;
   }
};
```

These are two class template instantiations:

```
Circle<float> c1;
Circle<double> c2;
```

These are equivalent to the above explicit structs.

C++ templates

Templates can be specialized:

Base template

```
template<typename T>
struct Circle
{
  T r;
  T area() const
  { return r*r*3.14; }
};
```

Specialization for `int`:

```
template<>
struct Circle<int>
{
  int r;
  double area() const
  { return (r*r)*3.14; }
};
```

C++ templates - SFINAE

Another example of specialization:

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

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

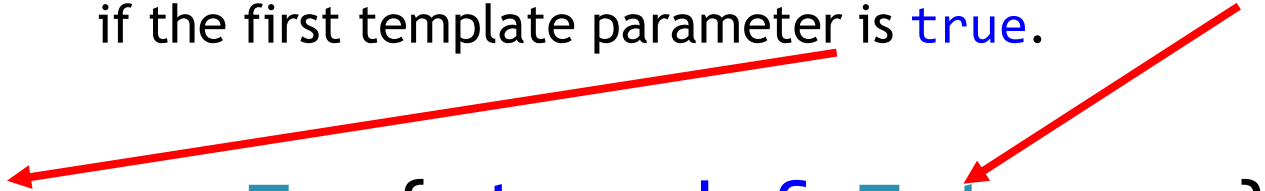
C++ templates - SFINAE

Another example of specialization:

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

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

This struct only possess a member type called: `type`, if the first template parameter is `true`.



C++ templates - SFINAE

SFINAE

C++ templates - SFINAE

SFINAE – The worst acronym in history

C++ templates - SFINAE

SFINAE – Substitution Failure Is Not An Error

C++ templates - SFINAE

SFINAE – Substitution Failure Is Not An Error

When a template instantiation is being resolved, multiple possible cases should be inspected for a possible match.

If some of the matches turn out to be ill-formed it will not be a compile error, it will be silently excluded from the set of possibilities.

C++ templates - SFINAE

Consider:

```
template <typename T,
          typename Checker = void>
struct Matrix
{...};
```

```
template <typename T>
struct Matrix<T, typename enable_if<is_noncommutative<T>::value>::type>
{...};
```

C++ templates - SFINAE

Consider:

```
template <typename T,
          typename Checker = void>
struct Matrix
{...};
```

`is_noncommutative` should be a similar template like `enable_if`, but it contains a `bool` member called `value` if its `T` argument is from a non-commutative algebra. The `Matrix` then can be specialized for that case.

```
template <typename T>
struct Matrix<T, typename enable_if<is_noncommutative<T>::value>::type>
{...};
```

C++ templates - SFINAE

SFINAE

Functions can be templates too,
but this interferes with overload-resolution...

Prepare for surprises.

C++ templates – expression SFINAE

Expression SFINAE in C++11

Things get more gets simpler

C++ templates - expression SFINAE

```
template<typename T> struct Circle
{
   T r;
   T area() const { return r*r*3.14; }
};

struct SomeShape
{
    double area;
};
```

C++ templates - expression SFINAE

```
template<typename T> struct Circle
{
   T r;
   T area() const { return r*r*3.14; }
};

struct SomeShape
{
   double area;
};
```

Here area is a function!

Here area is a variable!

C++ templates - expression SFINAE

```
template<typename S>
auto getArea( S shape )->decltype( shape.area() )
{
   return shape.area();
}


template<typename S>
auto getArea( S shape )->decltype( -shape.area )
{
   return shape.area;
}
```

C++ templates - expression SFINAE

```
template<typename S>
auto getArea( S shape )->decltype( shape.area() )
{
    return shape.area();
}
```

`decltype` is an operator since C++11, that calculates the type of the expression inside!

```
template<typename S>
auto getArea( S shape )->decltype( -shape.area )
{
    return shape.area;
}
```

C++ templates - expression SFINAE

```
template<typename S>
auto getArea( S shape )->decltype( shape.area() )
{
    return shape.area();
}
```

This is only valid for types having an area function

```
template<typename S>
auto getArea( S shape )->decltype( -shape.area )
{
    return shape.area;
}
```

This is only valid for types having an area variable

C++ templates - expression SFINAE

```
template<typename S>
auto getArea( S shape )->decltype( shape.area() )
{
    return shape.area();
}
```

getArea(Circle<float>{1.0})
binds here, since Circle has an area() function

```
template<typename S>
auto getArea( S shape )->decltype( -shape.area )
{
    return shape.area;
}
```

getArea(SomeShape{1.0})
binds here, since SomeShape has an area variable

C++ templates – expression SFINAE

With (expression) SFINAE we can specialize generic (functions) classes based on whether a given ***operation*** is valid on the argument types!

C++ templates – type_traits

The C++ Standard Library comes with many handy templates for checking properties of types.

C++ templates – TMP

Lets turn to template metaprogramming!

C++ templates – TMP

If the template level is a programming language...
What are the values?

The values are types:

```
struct x;
```

C++ templates - TMP

If the template level is a programming language...
What are the functions?

The functions are templates:

```
template<typename x> struct f;
template<typename x, typename y> struct f2;
template<int i>      struct g;
```

C++ templates - TMP

If the template level is a programming language...
What are the functions?

Example functions:

```
template<typename x>
struct id{ using result = x; };


template<int x> struct sq
{
  int value = x*x;
};
```

The name of the member, where we make the “return value” of the function available is a matter of **convention**!

C++ templates - TMP

If the template level is a programming language...
What are the functions?

Example functions:

```
template<typename x>
struct id{ using result = x; };
```

Equivalent to:

```
notype! id(notype! x){ return x; }
```

```
template<int x> struct sq
{
  int value = x*x;
};
```

Equivalent to:

```
int sq(int x){ return x*x; }
```

C++ templates - TMP

If the template level is a programming language...
What are the functions?

Example functions:

```
template<typename x>
struct id{ using result = x; };
```

```
template<int x> struct sq
{
  int value = x*x;
};
```

In practice, non-type templates are not preferred, because they cannot be abstracted over.

Things get much simpler, if we raise these values to the type level

C++ templates - TMP

```
template<int x>
struct Int
{
    static const int result = x;
};
```

Type level integer value

Specialization

```
template<typename x> struct sq{};
template<int i>      struct sq<Int<i>>
{ using result = Int<i*i>; };
```

C++ templates – TMP

Can’t we create a generic implementation?

C++ templates – TMP

Can’t we create a generic implementation?

Idea: `decltype` + operators

C++ templates - TMP

```
template<int x>
struct Int
{
    static const int result = x;
};

template<int i1, int i2>
auto operator*( Int<i1>, Int<i2> )
{
   return Int<i1*i2>();
}
```

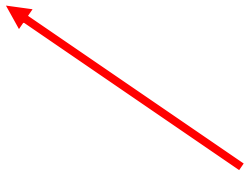
Type level integer value

Note: variable names can be omitted

C++ templates - TMP

The generic implementation:

```
template<typename x> struct sq
{
 using result = decltype (x() * x ());
};
```



This creates an instance of type x

C++ templates – TMP

But wait...

Why this needs to be a class template?

C++ templates - TMP

```
template< typename X >
auto sq(X x){ return x*x; }
```

This function now works for both values, and both template level stuff, like `Int<4>`:

```
some_class< decltype( sq( Int<4>() ) ) > c;
```

C++ templates - TMP

```
template< typename X >
auto sq(X x){ return x*x; }
```

We first create a value of the type `Int`

Use all our operators and generic functions inside the `decltype`

After `decltype`, we get back to type level, and can use the type of the result in templates, utilize speicalization, etc.

```
some_class< decltype( sq( Int<4>() ) ) > c;
```

C++ templates - TMP

```
template< typename X >
auto sq(X x){ return x*x; }


some_class< decltype( sq( Int<4>() ) ) > c;
```

It will be equivalent to

```
some_class< Int<16> > c;
```

C++ templates – TMP

More metaprogramming:

Type level lists (or arrays, as you like)

C++ templates - TMP

Type level lists:

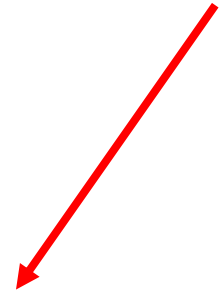
```
template<typename... elems> struct List;

using xs = List<Int<1>, Int<2>, Int<3>>;
```

C++ templates - TMP

Type level lists:

Since C++11 variadic templates are allowed, that can represent 0 or more template arguments of any type!



```
template<typename... elems> struct List;

using xs = List<Int<1>, Int<2>, Int<3>>;
```

C++ templates – TMP

Ok, but how we get elements out?

C++ templates - TMP

Ok, but how we get elements out?

First element:

```
template<typename... elems> struct List;

template<typename L> struct Fst;

template<typename A0, typename... As>
struct Fst<List<A0, As...>>{ using result = A0; };

using x0 = typename Fst<List<Int<1>, Int<2>, Int<3>>>::result;
```

Usual trick: break the pack into first and rest!

The list will be pattern matched against `A0` and `As...`

Will be `Int<1>`

C++ templates – TMP

Ok, but how we get the other elements out?

Recursion!

C++ templates – TMP

```
//Helper struct
template<typename Icurrent, typename Igoal, typename L> struct Nth_impl;


template<typename Icurrent, typename Igoal, typename A, typename... As>
struct Nth_impl<Icurrent, Igoal, List<A, As...>> //The induction step
{
        using result = typename Nth_impl<Int<Icurrent::result+1>,
                                         Igoal, List<As...>>::result;
};
template<typename I, typename A, typename... As> //Closing step of recursion
struct Nth_impl<I, I, List<A, As...>>  { using result = A; };


template<typename L, typename I> //End-user alias
using Nth = typename Nth_impl<Int<0>, I, L>::result;
```

C++ templates - TMP

```
//Helper struct
template<typename Icurrent, typename Igoal, typename L> struct Nth_impl;


template<typename Icurrent, typename Igoal, typename A, typename... As>
struct Nth_impl<Icurrent, Igoal, List<A, As...>> //The induction step
{
        using result = typename Nth_impl<Int<Icurrent::result+1>,
                                         Igoal, List<As...>>::result;
};

template<typename I, typename A, typename... As> //Closing step of recursion
struct Nth_impl<I, I, List<A, As...>>  { using result = A; };
```

The recursion ends, when `Icurrent == Igoal`

This is just the same as `Fst` we just wrote!

C++ templates - TMP

Lets write this out on an example:

```
L = List<Int<1>, Int<2>, Int<3>>, Igoal = Int<2>
```

A, As...

Step 1:

```
struct Nth_impl<Int<0>, Int<2>, List<Int<1>, Int<2>, Int<3>>>
{
 using result =
 typename Nth_impl<Int<1>, Int<2>, List<Int<2>, Int<3>> >::result;
};
```

This instantiates step 2.:

A, As...

Step 2:

```
struct Nth_impl<Int<1>, Int<2>, List<Int<2>, Int<3>>>
{
 using result =
 typename Nth_impl<Int<2>, Int<2>, List<Int<3>> >::result;
};
```

The recursion ends, because `Icurrent == Igoal`

C++ templates - TMP

Lets write this out on an example:

```
L = List<Int<1>, Int<2>, Int<3>>, Igoal = Int<2>
```

Step 1:

```
struct Nth_impl<Int<0>, Int<2>, List<Int<1>, Int<2>, Int<3>>>
{
 using result =
 typename Nth_impl<Int<1>, Int<2>, List<Int<2>, Int<3>> >::result;
};
```

This instantiates step 2.:

Step 2:

```
struct Nth_impl<Int<1>, Int<2>, List<Int<2>, Int<3>>>
{
 using result =
 typename Nth_impl<Int<2>, Int<2>, List<Int<3>> >::result;
};
```

The recursion ends, because `Icurrent == Igoal`

C++ templates - TMP

Lets write this out on an example:

```
L = List<Int<1>, Int<2>, Int<3>>, Igoal = Int<2>
```

Step 1:

```
struct Nth_impl<Int<0>, Int<2>, List<Int<1>, Int<2>, Int<3>>>
{
 using result =
 typename Nth_impl<Int<1>, Int<2>, List<Int<2>, Int<3>> >::result;
};
```

A, As...

This instantiates step 2.:

Step 2:

```
struct Nth_impl<Int<1>, Int<2>, List<Int<2>, Int<3>>>
{
 using result =
 typename Nth_impl<Int<2>, Int<2>, List<Int<3>> >::result;
};
```

A, As...

The recursion ends, because `Icurrent == Igoal`

C++ templates - TMP

Ok, but how we get the other elements out?

```
using xn = typename Nth< List<Int<1>, Int<2>, Int<3>>, Int<n> >::result;
```

We can again hide it behind an operator:

```
template<typename... L> struct List
{
  template<int n>
  auto operator[]( Int<n> )
  { return Nth<List<L...>,  Int<n> >(); }
};
```

C++ templates – TMP

Okay, we have values, functions, lists, what can we do with them?

C++ templates – TMP

Typical applications of template-metaprogramming:

- Dealing with structures and functions operating on variadic (but compile time known) number of arguments:

  implementing `std::tuple`

  checking type traits on collections of types
  and other type level algorithms

  creating a type-safe version of
  `printf("value is %i", x)`

C++ templates – TMP

Typical applications of template-metaprogramming:

- Explicit inlined repetition of parametric tasks

you might use a loop, but you either want to be sure, that it will be unrolled, or

you need the result at compile time!
(e.g. to use it in a template, c.f. integer sequences, partitions, combinatorics, precalculated exact rational expressions...)

C++ templates – TMP

Typical applications of template-metaprogramming:

- Concrete examples:

  Runge-Kutta steppers ← See the advanced samples
  for arbitrary stages

  Finite-Difference stencils
  for multiple dimensions, arbitrary order

C++ templates – TMP

Another major application:

Expression templates!

C++ templates – TMP

Idea:

What if we want to do something with the *structure* of a computation?

C++ templates – TMP

Idea:

What if we want to do something with the *structure* of a computation?

Lazy evaluation

C++ templates - TMP

Expression templates - simple example

```
template<typename T>
struct Variable
{
  T val;
  auto operator()() const { return val; }
};

template<typename T>
auto var( T const t ){ return Variable<T>{t}; }

template<typename T>
bool is_expression( Variable<T> ){ return true; }
```

C++ templates - TMP

Expression templates - simple example

```
template<typename T>
struct Variable
{
  T val;
  auto operator()() const { return val; }
};
```

Simple struct to hold any type of value, and provide it when () is called

```
template<typename T>
auto var( T const t ){ return Variable<T>{t}; }
```

Helper function to create a `Variable<T>` (infers the template type)

```
template<typename T>
bool is_expression( Variable<T> ){ return true; }
```

Helper function that says, `Variable<T>` is an expression (see later)

C++ templates - TMP

Expression templates - simple example

```
template<typename Op, typename Left, typename Right> struct Expression
{
  Op op; Left left; Right right;
  auto operator()() const { return op(left(), right()); }
};

template<typename Op, typename Left, typename Right>
auto expr( Op const& op, Left const& l, Right const& r )
{ return Expression<Op, Left, Right>{op, l, r}; }

template<typename... T> bool is_expression( Expression<T...> ){ return true; }
```

C++ templates - TMP

Expression templates - simple example

A struct to hold a binary function, and two (callable) types, when () is called, op is evaluated on the results of left() and right()

```
template<typename Op, typename Left, typename Right> struct Expression
{
  Op op; Left left; Right right;
  auto operator()() const { return op(left(), right()); }
};
```

Helper function to create an Expression (infers the template types)

```
template<typename Op, typename Left, typename Right>
auto expr( Op const& op, Left const& l, Right const& r )
{ return Expression<Op, Left, Right>{op, l, r}; }
```

Helper function that says, Expression<T...> is an expression (see later)

```
template<typename... T> bool is_expression( Expression<T...> ){ return true; }
```

C++ templates - TMP

Expression templates - simple example

And now an addition operator:
here we use the SFINAE with the `is_expression` helper functions
to restrict this operator only to our types defined above.

`std::declval` is a helper function to return a `T&&` to any type.
Only to be used inside a `decltype`

```
template<typename Left,
         typename Right,
         typename Rq1 = decltype(is_expression(std::declval<Left>())),
         typename Rq2 = decltype(is_expression(std::declval<Right>()))>
auto operator+( Left const& l, Right const& r )
{
   return expr([](auto x, auto y){ return x+y; }, l, r);
}
```

C++ templates – TMP

Expression templates – simple example

And now a multiplication:

```
template<typename Left,
         typename Right,
         typename Rq1 = decltype(is_expression(std::declval<Left>())),
         typename Rq2 = decltype(is_expression(std::declval<Right>()))>
auto operator*( Left const& l, Right const& r )
{
   return expr([](auto x, auto y){ return x*y; }, l, r);
}
```

C++ templates - TMP

Expression templates - simple example

And NOW if we write:

```
auto c = (var(2.0) + var(4.0)) * var(5.0);
std::cout << "type of c is: " << typeid(c).name() << " value is: "
          << c() << std::endl;
```

The output is (reformatted for readability):

This is our multiplication lambda inside operator*

This is our addition lambda inside operator+

```
type of c is:
struct Expression< class <lambda_9a68...>,
                  struct Expression< class <lambda_29b5...>,
                                     struct Variable<double>,
                                     struct Variable<double> >,
                  struct Variable<double> >
value is: 30.000000000000000
```

C++ templates – TMP

Why is it good to have the expression trees?

We can do whatever we want *before* evaluation.

Example: see the simple symbolic differentiation example,
it includes the printer of the expressions to the console

Symbolic optimizations on such trees are very common.

C++ templates – TMP

Why is it good to have the expression trees?

A similar method that is using the CRTP pattern
is widely used for handling vector and matrix expressions.

It builds a similar tree, but it uses `operator[]`
as the common interface function to extract values

while avoiding the allocations of temporaries!!!

C++ templates – TMP

While some of these metaprogramming tricks are known since 20 years, they did not make into the standard library.

However, the boost library collection heavy uses such techniques.

In STL 2 such expression template based constructs will be used more often.