Slide 1

Template Metaprogramming

Metaprograms are templates which generate code/programs at compile-time.

Think of a metaprogram as a program written in the *template language*, and the program generates code in the C++ language.

The metaprograms are often short and can be used to generate efficient code, like unroll loops.

Unfortunately, they can be difficult to understand.

Slide 2

Example1: The Power of 3

The metaprogram `Pow3<N>` works by recursive template instantiation using the rules:

1. $3^N = 3 * 3^{N-1}$
2. $3^0 = 1$

```
// primary template to compute 3 to the Nth
template <int N>
class Pow3 {
public:
  enum { result = 3 * Pow3<N-1>::result };
};
```

Slide 3

Example1: The Power of 3 ...

Notice! the public variable `result` is an `enum`, which is a true constant (not an lvalue)

It can also be implemented as a `static int` (but then it is an lvalue).

```
// full specialization to end the recursion
template <>
class Pow3<0> {
public:
  enum { result = 1 };
};
```

Slide 4

Use of Pow3

```
#include <iostream>
#include "pow3.h"

int main()
{
  std::cout << "Pow3<4>::result = "
            <<  Pow3<4>::result << std::endl;

  return 0;
}
```

Slide 5

Use of Pow3 . . .

First, the compiler instantiates `Pow3<4>`. The result is

```
3 * Pow3<3>::result
```

Then `Pow3<3>` is instantiated etc. The results are

```
3 * 3 * Pow3<2>::result
3 * 3 * 3 * Pow3<1>::result
3 * 3 * 3 * 3 * Pow3<0>::result
3 * 3 * 3 * 3 * 1
81
```

This value `81` is stored in the public variable `Pow3<4>::result`.

Slide 6

Use of Pow3 . . .

The compiler will see the fully instantiated template `Pow3<4>` as

```
class Pow3<4> {
public:
  enum { result = 3 * 3 * 3 * 3 * 1 };
};
```

and it will evaluate the constant expression `3 * 3 * 3 * 3 * 1` at compile time and assign `81` to the variable `result`.

Slide 7

Use of Pow3 . . .

After the compilation, the code generated for class `Pow3<4>` will be the same as if it had been declared as

```
class Pow3<4> {
public:
  enum { result = 81 };
};
```

So, when the expression `Pow3<4>::result` is used in a program, no computations are performed.

Slide 8

Example2: The Integer Square Root

The metaprogram `Sqrt<N>` computes the integer square root of `N` by performing a binary search at compile-time.

```
template <int N, int LO=1, int HI=N>
class Sqrt {
public:
  enum { mid = (LO + HI + 1) / 2 };
  enum { result = (N < mid * mid) ?
                   Sqrt<N, LO, mid-1>::result
                 : Sqrt<N, mid, HI>::result };
};
```

Slide 9

Example2: The Integer Square Root ...

The recursion is stopped by declaring a partial specialization of `Sqrt` for the case where `LO = HI`.
In this case `LO = HI = M` the result (the square root) is equal to `M`

```
template <int N, int M>
class Sqrt<N, M, M> {
public:
  enum { result = M };
};
```

Slide 10

Use of Sqrt

```
#include <iostream>
#include "sqrt1.h"
int main()
{
  std::cout << "Sqrt<9>::result = "
            <<  Sqrt<9>::result << std::endl;
  return 0;
}
```

Notice! There is only one template parameter, because the other two have default values.

Slide 11

Use of Sqrt ...

The expression `Sqrt<9>` is expanded to

```
Sqrt<9,1,9>::result
```

Inside the metaprogram the following happens

```
mid = (1 + 9 + 1) / 2 = 5
result = (9 < 5 * 5) ? Sqrt<9, 1, 4>::result
                     : Sqrt<9, 5, 9>::result
       = (9 < 25)    ? Sqrt<9, 1, 4>::result
                     : Sqrt<9, 5, 9>::result
       = Sqrt<9, 1, 4>::result
```

Slide 12

Use of Sqrt ...

The expression `Sqrt<9, 1, 4>::result` is expanded to

```
mid = (1 + 4 + 1) / 2 = 3
result = (9 < 3 * 3) ? Sqrt<9, 1, 2>::result
                     : Sqrt<9, 3, 4>::result
       = (9 < 9)     ? Sqrt<9, 1, 2>::result
                     : Sqrt<9, 3, 4>::result
       = Sqrt<9, 3, 4>::result
```

Slide 13

Use of Sqrt ...

The expression `Sqrt<9, 3, 4>::result` is expanded to

```
mid = (3 + 4 + 1) / 2 = 4
result = (9 < 4 * 4) ? Sqrt<9, 3, 3>::result
                     : Sqrt<9, 4, 4>::result
       = (9 < 16)    ? Sqrt<9, 3, 3>::result
                     : Sqrt<9, 4, 4>::result
       = Sqrt<9, 3, 3>::result
       = 3
```

and `Sqrt<9, 3, 3>::result` ends the recursion by assigning `3` to its public variable `result`.

Slide 14

Use of Sqrt ...

When a compiler evaluates the expression

```
(9 < 5 * 5) ? Sqrt<9, 1, 4>::result
            : Sqrt<9, 5, 9>::result
```

it instantiates both `Sqrt<9, 1, 4>::result` and `Sqrt<9, 5, 9>::result`.

The code attempts to access the member `result`, so all members are instantiated.

This is very time consuming uses a lot of memory!

Slide 15

IfThenElse

To avoid the number of template instantiations, a new metaprogram `IfThenElse` is implemented

```
template <bool C, typename Ta, typename Tb>
class IfThenElse;
```

with partial specializations for the boolean expression `C`.

If `C == true` the metaprogram yields its second template argument, and if `C == false` it yields its third template argument.

Slide 16

IfThenElse . . .

```
template <typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb> {
public:
  typedef Ta ResultT;
};

template <typename Ta, typename Tb>
class IfThenElse<false, Ta, Tb> {
public:
  typedef Tb ResultT;
};
```

Slide 17

New Sqrt

```
template <int N, int LO=1, HI=N>
class Sqrt {
public:
  enum { mid = (LO + HI + 1) / 2 };
  typedef
    typename
    IfThenElse<(N < mid * mid),
               Sqrt<N, LO, mid-1>,
               Sqrt<N, mid, HI> >::ResultT SubT;
  enum { result = SubT::result };
};
```

Slide 18

New Sqrt ...

The only thing the metaprogram `IfThenElse` does, is to define a `typedef` for a template class instance.

Doing a `typedef` definition does not instantiate the template class in question.

When using the `IfThenElse` metaprogram neither its second argument nor its third argument is fully instantiated.

Slide 19

New Sqrt ...

Finally, to stop the recursive instantiations, the following partial specialization is needed

```
template <int N, int M>
class Sqrt<N, M, M> {
public:
  enum { result = M }
};
```

This version of the `Sqrt` metaprogram reduces the number of instantiations from approximately `2N` to $\log_2$(`N`).

Slide 20

Induction Variables

Previously, the metaprogram `Sqrt` was implemented in a recursive way.

A more "naive iterative algorithm" can be formulated as follows:

To compute the square root of a given value `N`, write a loop which iterates a variable `I` until its square is equal to or greater than `N`.

Slide 21

Induction Variables ...

In ordinary C++ it could look like

```
...
int I = 1;
while (I * I < N) { ++I; }
// now I contains the integer square root of N
...
```

Written as a metaprogram, the loop must be formulated in a recursive way.

Slide 22

Induction Variables ...

A metaprogram for computing the square root of N in an iterative way could look like

```
template <int N, int I=1>
class Sqrt {
public:
  enum { result = (I * I < N) ?
                     Sqrt<N, I+1>::result
                   : I };
};
```

Slide 23

Induction Variables ...

With the partial specialization to end the recursion

```
template <int I>
class Sqrt<I, I> {
public:
  enum { result = I };
};
```

This partial specialization is necessary, because of the template instantiations.

Slide 24

Induction Variables ...

Computing `Sqrt<4>` goes as follows:

```
Step 1:
  result = (1 * 1 < 4) ?
             Sqrt<4, 2>::result
           : 1
Step 2:
  result = (1 * 1 < 4) ?
             (2 * 2 < 4) ?
               Sqrt<4, 3>::result
             : 2
           : 1
```


Slide 25

Induction Variables ...

```
Step 3:
  result = (1 * 1 < 4) ?
             (2 * 2 < 4) ?
               (3 * 3 < 4) ?
                 Sqrt<4, 4>::result
               : 3
             : 2
           : 1
```

Slide 26

Induction Variables ...

```
Step 4:
  result = (1 * 1 < 4) ?
             (2 * 2 < 4) ?
               (3 * 3 < 4) ?
                 4
               : 3
             : 2
           : 1
```

The result is found in `Step 2`, but the instantiation continues until a specialization is found.

Slide 27

Induction Variables . . .

The problem can be solved using the metaprogram `IfThenElse`, which works on types.

But then a type must be defined, which can contain the value of `I`

```
template <int I>
class Value {
public:
  enum { result = I };
};
```

Slide 28

Induction Variables . . .

```
template <int N, int I=1>
class Sqrt {
public:
  typedef
    typename IfThenElse<(I * I < N),
                        Sqrt<N, I+1>,
                        Value<I>
             >::ResultT SubT;
  enum { result = SubT::result };
};
```

Slide 29

Unroll Loops

Usually, computations like dot products of vectors are evaluated using loops:

```
template <typename T>
T dot_product(int dim, T* a, T* b)
{
  T result = T();
  for (int i = 0; i < dim; ++i) {
    result += a[i] * b[i];
  }
  return result;
}
```

Slide 30

Unroll Loops ...

The template function `dot_product` is called as follows:

```
int main()
{
  int a[3] = {1, 2, 3};
  int b[3] = {5, 6, 7};

  std::cout << "dot_product(3, a, b) = "
            <<  dot_product(3, a, b) << std ::endl;
  return 0;
}
```

Slide 31

Unroll Loops . . .

Loops are time consuming, so time can be saved if they are unrolled.

```
template <int DIM, typename T>
class DotProduct {
public:
  static T result(T* a, T* b) {
    return
      *a * *b
        + DotProduct<DIM-1,T>::result(a+1,b+1);
  }
};
```

Slide 32

Unroll Loops . . .

Using the partial specialization as stop criterion for the recursion

```
template <typename T>
class DotProduct<1, T> {
public:
  static T result(T* a, T* b) {
    return *a * *b;
  }
};
```

Slide 33

Unroll Loops . . .

The following template function might be convenient to make this implementation look like the previous one.

```
// convenience function
template <int DIM, typename T>
T dot_product(T* a, T* b)
{
  return DotProduct<DIM,T>::result(a, b);
}
```

Slide 34

Unroll Loops . . .

The new template function `dot_product` is called as follows:

```
int main()
{
  int a[3] = {1, 2, 3};
  int b[3] = {5, 6, 7};

  std::cout << "dot_product<3>(a, b) = "
            <<  dot_product<3>(a, b) << std ::endl;
  return 0;
}
```

Slide 35

Unroll Loops . . .

The expression `dot_product<3>(a, b)` results in the following instantiations

```
DotProduct<3,int>::result(a, b)
= *a * *b + DotProduct<2,int>::result(a+1, b+1)
= *a * *b + *(a+1) * *(b+1)
    + DotProduct<1,int>::result(a+2, b+2)
= *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2)
```