Part XI

Advanced Templating

Outline

Instantiation models

- See `https://gcc.gnu.org/onlinedocs/gcc/Template-Instantiation.html` for full technical details.
- If you are using `gcc` or similar compilers `icpc`, `clang++` then the following should hold:
- Template code is instantiated when the compiler encounters a use of a templated entity with particular template parameters (either explicit or deduced).
- The compiler will only instantiate code which is accessible to it at that point.
- Thus, you typically need to declare all templated code in a header file.
- When your templated class or function is used within a `.C` file, the compiler will be able to instantiate all the code visible to it (i.e. further up the pre-processed source-file)

Instantiation models - example

`sum.hpp`:

```
template<typename T>
T sum(const std::vector<T>& v);
#include "sumTemplates.hpp"
```

`sumTemplates.hpp`:

```
template<typename T>
T sum(const std::vector<T>& v){
  T total = (T)0;
  for(const T& i : v){
    total += i;
  }
}
```

`sum.cpp`:

```
#include "sum.hpp"
int main(void){
  std::vector<int> values;
  int f = sum(values);
}
```

Instantiation models - example

- When it encounters `sum(values)` the compiler instantiates and compiles the templated function `sum`.
- The compiled code is placed into the object file `sum.o`.
- This is done for each `.cpp` file that refers to `sum(const std::vector<int>&)`
- Of course, `std::vector` is itself a template, so the compiler generates code for `std::vector<int>::operator[]` etc. for each object file as well.
- With only the declaration of `sum()` from `sum.hpp`, but not its definition from `sumTemplates.hpp`, the code will compile, but not link.

Instantiation models - problems

- There are some drawbacks to the above approach:
  - A templated function/class may be compiled multiple times (once for each object file it is contained in), resulting in larger object files, and longer compile times than necessary. However, multiple definitions are discarded by the linker.
  - If you change a templated function/class, then every `.C` file it is included in must be recompiled. (If not, then multiple, *non-matching* definitions of functions/classes could be found by the linker. Murphy's law says that the wrong ones will be discarded, resulting in hours of confusion.)
- These drawbacks are not a problem in practice; disk-space and compile-time are sufficiently cheap not to matter too much.

Instantiation models - explicit instantiation

- You might wonder whether we can avoid the implicit instantiation above.
- We can to some extent; if we have `sum.hpp`, but not `sumTemplates.hpp` in the above, then the compiler can compile `sum.cpp` as it knows the function signature of `sum`.
- However, we must then ensure that a version of `sum<int>` is compiled somewhere.
- We could have: `sumInstantiate.cpp`:

```
#include "sumTemplates.hpp"
template int sum(const std::vector<int>&);
```

- In general, this reduces compile time and space for object files.
- However, it also means that we have to add an extra line to `sumInstantiate.cpp` for every type for which we need to instantiate `sum()`.

Instantiation models

- Explicit instantiation requires maintenance overhead in simple cases. In complex cases it becomes infeasible.
- Further, explicit instantiation usually means that small templated functions cannot be inlined at the point of calling. This may make a substantial difference to how well the code can be optimized.
- You are strongly encouraged to use the implicit instantiation approach unless you know what you are doing.

Outline

Expression Templates in detail

- In the original C++ lectures I explained some of the structure of expression templates.
- For masochists, here are the full details of what goes on.
- Full details can be found in Vandevoorde and Josuttis.

A reminder

We want the following:

```
Vector a(10), b(10), c(10);
c = 2.3*a + 4.5*b + a*b; // Assume elt-wise multiplication
```

to be evaluated without creating intermediate temporary variables for the subexpressions:

```
Vector tmp1(10) = 2.3*a;
Vector tmp2(10) = 4.5*b;
Vector tmp3(10) = tmp1 + tmp2;
Vector tmp4(10) = a*b;
Vector tmp5(10) = tmp3 + tmp4;
c = tmp5;
```

Basic vector

Suppose we have a simple `Vector` class, with fixed size known at compile time.

```
template<int SIZE>
class Vector{
public:
  Vector(){}
  Vector(const Vector& a){
    for(std::size_t i=0 ; i < SIZE ; i++){
      m_data[i] = a[i];
    }
  }
  Vector& operator=(const Vector& a){
    for(std::size_t i=0 ; i < SIZE ; i++){
      m_data[i] = a[i];
    }
    return *this;
  }
  double operator[](std::size_t i)const{
    return m_data[i];
  }
  double& operator[](std::size_t i){
    return m_data[i];
  }
private:
  double m_data[SIZE];
};
```

Vector expression

- We want to create a copy-constructor from a Vector expression:

```
template<int SIZE>
template<typename VectorExpression>
Vector<SIZE>& Vector<SIZE>::operator=(const VectorExpression& v)
  for(size_t i=0 ; i < SIZE ; i++){
    m_data[i] = v[i];
  }
}
```

- This has two drawbacks:
  1. We have no guarantee that the `VectorExpression` type has the correct `SIZE`.
  2. This could apply to any type, including a `std::vector`, a `std::valarray`, etc. (this *could* be an advantage).
- We create a general `VectorExpr` type instead.

Vector expression type

- From the previous copy-constructor, a `VectorExpr` just needs to have an `operator[]` function.
- A `VectorExpr` may not hold any data as such; the idea is that we do not create extra storage for the intermediate sub-expressions.
- So, its data storage is a generic type that only has to implement a `operator[]`.
- (This feels as though we have the same problem as with the copy-constructor before, but not quite; `VectorExpr` is under our control, and we only allow `VectorExpr` objects to be created by our functions.
- So, we have:

```
template<int SIZE, typename InternalData> class VectorExpr;
```

Vector expression type

```
template<int SIZE, typename InternalData>
class VectorExpr{
public:
  VectorExpr(const InternalData& d) : m_data(d) { }

  double operator[](size_t i){
    return m_data[i];
  }
private:
  InternalData m_data;
};
```

- So, what is `InternalData`, and how do we construct a `VectorExpr`?
- We now look at the problem from the other end...

Vector expression construction

- What is the result of `a + b` or `Vector<SIZE> + Vector<SIZE>` ?
- We do not want to evaluate the result until the copy-constructor for a `Vector` is called.
- The copy-constructor calls the `operator[]`, so we create an object that stores the operand `Vector`s internally, and only evaluates the sum of them in `operator[]`
- For generality, we want to be able to sum multi-term expressions, so the operands may themselves be vector-expressions, not just `Vector`s.

Vector expression construction

```
template<int SIZE, typename Op1, typename Op2>
class VectorAdd{
public:
  VectorAdd(const Op1& a, const Op2& b) : op1(a), op2(b) { }

  double operator[](size_t i)const{
    return op1[i] + op2[i];
  }
private:
  Op1 op1;
  Op2 op2;
};
```

- Again, the operand types only need to have an `operator[]` implemented.

Vector expression construction

- So, how do we construct a `VectorAdd` object? It needs to be the result of an `operator+`:
- The `operator+` needs to take two general operands: `VectorExpr<SIZE, InternalData1>` and `VectorExpr<SIZE, InternalData2>`
- It then constructs a `VectorAdd` object with these as the contained data.
- Sketchily:

```
operator+(const Op1& op1, const Op2& op2){
return VectorAdd<SIZE, Op1, Op2>(op1, op2);
}
```

- However, this will overload `+` for any two possible operands, of any type...

Vector expression construction

- We want to restrict the `operator+` to vector expressions we control, i.e. not `a + std::vector<int>(10)`.
- This is why we created `VectorExpr`.
- So, the `operator+` acts on two `VectorExpr` objects, which might have any internal storage.
- In order to be part of large expressions, it must also return a `VectorExpr`:

```
template<int SIZE, typename IData1, typename IData2>
VectorExpr<SIZE,VectorAdd<SIZE, IData1,IData2>>
operator+(const VectorExpr<SIZE, IData1>& op1,
          const VectorExpr<SIZE, IData2>& op2){
  return VectorExpr<SIZE,
                    VectorAdd<SIZE, IData1,IData2>>
         (VectorAdd<SIZE, IData1, IData2>(op1, op2));
}
```

Vector expression construction

- Finally, we need to ensure that a basic `Vector` can be seen as a `VectorExpr`.
- Otherwise, the `operator+` will not apply to it correctly.
- This is actually slightly more complex than it first appears; even if we implement a conversion from a `Vector` to a `VectorExpr`, the compiler does not perform the conversion we might expect.
- We have to change our classes as follows:

  `Vector<SIZE>` $\mapsto$ `SimpleVector<SIZE>`

  `VectorExpr<SIZE, InternalData>` $\mapsto$

  `Vector<SIZE, InternalData = SimpleVector<SIZE>>`
- So, we usually use a `Vector` type in our code, which by default has a `SimpleVector` as its storage.
- This makes the `operator+` a little more complicated, because we now have to wrap the internal data-types in a `Vector`.

Vector assignment

- We now only need to create the assignment from a vector-expression to a vector:

```
template<int SIZE, typename IData>
template<typename IData2>
const Vector<SIZE, IData>&
Vector<SIZE, IData>::operator=
(const Vector<SIZE, IData2>& v)const{
  for(size_t i=0 ; i < SIZE ; i++){
    m_data[i] = v[i];
  }
  return *this;
}
```

- This ensures that a `Vector` can only be constructed from an `Vector`-expression of the same size.
- See `Examples/expressionTemplates.C` for the full code.

More operations

- We also want to have more functionality for our expressions.
- Implementing `VectorMultiply`, `VectorSubtract`, `VectorDivide` is easy:

```
#define VectorBinaryOp(Name, Op)       \
class Vector##Name{                    \
  double operator[](size_t i)const{   \
    return Op1[i] Op Op2[i];           \
  }                                    \
};

VectorBinaryOp(Add, +)
VectorBinaryOp(Multiply, *)
```

- Much code omitted above: constructor, member data, etc.

Scalars

- Applying scalars to this is a little more complicated.
- We want to overload `operator+(Vector<...> a, double b)`.
- This needs to return `Vector<VectorAdd<A, B>>`. What is `B`?
- We can create a simple class `Scalar` that behaves like a `Vector`:

```
class Scalar{
public:
  Scalar(double v) : m_value(v){}
  double operator[](size_t)const{return m_value;}
private:
  double m_value;
}
```

- It's like a `Vector` that has all its elements equal to `m_value`.
- This means that we can construct a `VectorAdd` that has a `Scalar` as an internal data-type.
- Otherwise we would have to create a separate `VectorAdd` class for each of `2.0 + v` and `v + 2.0`.

Scalars ctd

- Thus, we have to overload `operator+` again:

```
template<int SIZE, typename InternalData1>
Vector<SIZE,VectorAdd<SIZE, Vector<SIZE,InternalData1>,Scalar>>
operator+(const Vector<SIZE, InternalData1>& op1,
          const double& op2){
  return Vector<SIZE,
    VectorAdd<SIZE, Vector<SIZE, InternalData1>, Scalar>>
    (VectorAdd<SIZE, Vector<SIZE,InternalData1>, Scalar>(op1,
    op2));
}
```

- Note that this only allows `v + 3.0`. We need another very similar overload to support `3.0 + v`.
- However, this is less work than three versions of `VectorAdd<>`.

Unary operators and functions

- For a fully-fledged `Vector` class you need to overload unary + and unary `-` as well.
- It is simple to extend the binary operator macros above.
- You may also wish to have `sin(v)` and `atan2(v, w)` on an element-wise basis.
- This is also straightforward, again using very similar macros to those above.

How does this work?

- Once the full object encapsulating the expression is formed, all the compiler has to do is optimize the `operator[]` call.
- Since the `operator[]` functions are all very simple, the compiler can inline every containing function.
- Thus, the `operator=` function is essentially:

```
for(size_t i=0 ; i < SIZE ; i++){
  m_data[i] = a.m_data[i] + 3*b.m_data[i] + c.m_data[i] *
    d.m_data[i];
}
```

- One additional point: the `op1` and `op2` members of `VectorAdd` should be `const Op1&` for everything except a `Scalar`.
- This avoids copy-constructors as well, and may allow the compiler to make evern better optimizations.

How does the compiler cope?

- The overall type of an expression is very long and complicated.
- The C++ standard requires that a compiler support up to 1024 nested template instantiations.
- Since each extra operator ends up creating an extra two levels of nesting, this suggests a maximum of around 512 terms.
- 512 terms should be enough for anyone...
- Note that some of the preceding could be simplified by judicious use of `auto` and `decltype`. However, not making use of these allows you to appreciate what the compiler actually has to do, under the hood.

Extensions

Other features you could imagine extending this with would be:

- Generic data-type (e.g. `bool/float/int/double/complex`) - requires some care with type-promotion (`std::common_type` may be useful).
- Extra element-wise functions, such as `minmod()`.
- Logical operator and bitwise overloads for `&`, `&&` etc.
- Conditional operations. Since the ternary operator `? :` cannot be overloaded, you will have to create a function `if_()`.
- Some kind of CUDA-kernel back-end. Only the `operator=` needs to have a `__global__` kernel launch; so long as the `operator[]` are `__host__ __device__` and the data is all on the GPU, it should work.

Examples

Complex examples of expression templates include:

- GiNaC (`https://www.ginac.de/`) - a symbolic manipulation package implemented in C++
- FTensor (`https://bitbucket.org/wlandry/ftensor/`) - implementation of tensor manipulation and summation convention in C++
- Boost::Yap (`https://github.com/boostorg/yap`) - allows you to add expression template semantics to existing classes.