

Advanced C++ Template Techniques: Introduction to Meta-Programming for Scientific Computing



Dr Conrad Sanderson
Senior Research Scientist
NICTA Queensland Laboratory

conradsand@ieee.org

Version 1.0 (2009-10-25)

Templates were originally designed to reduce duplication of code:

```
// traditional approach: define a function for each type
float distance(float a1, float a2, float b1, float b2)
{
    float tmp1 = a1 - b1;
    float tmp2 = a2 - b2;
    return std::sqrt( tmp1*tmp1 + tmp2*tmp2 );
}
```

```
double distance(double a1, double a2, double b1, double b2)
{
    double tmp1 = a1 - b1;
    double tmp2 = a2 - b2;
    return std::sqrt( tmp1*tmp1 + tmp2*tmp2 );
}
```

```
// OR, use a template -- can handle both float and double
template <typename T>
T distance(T a1, T a2, T b1, T b2)
{
    T tmp1 = a1 - b1;
    T tmp2 = a2 - b2;
    return std::sqrt( tmp1*tmp1 + tmp2*tmp2 );
}
```

- Traditional way of computing the factorial (“n!”, e.g. “4! = 4 * 3 * 2 * 1”)

```
int factorial(int n)
{
    if (n==0)                // terminating condition
        return 1;
    else
        return n * factorial(n-1); // recursive call to factorial()
}

void user_function()
{
    int x = factorial(4);     // 4 * 3 * 2 * 1 * 1 = 24
    ...
}
```

- Problem? Slow! Value “4” is known at compile time
- We can also use templates for “meta-programming”
- Compiler can be forced to evaluate function-like constructs at *compile time*

- Computing the factorial function through template meta-programming (“n!”, e.g. “4! = 4 * 3 * 2 * 1”)

```
template <int N>
struct Factorial
{
    static const int value = N * Factorial<N-1>::value; // recursive!
};

template <>
struct Factorial<0> // template specialisation
                   // required for terminating condition
{
    static const int value = 1;
};

void user_function()
{
    int x = Factorial<4>::value; // 24, known at compile time
    ...
}
```

- Traditional method: compute factorial at *run time*
- Meta-programming: compute factorial at *compile time*
 - smaller code
 - faster execution

- So how useful is this in real life ?
- Say we want to convert some Matlab code to C++
- Need a matrix library
- Following a traditional approach, we could define a simple matrix class:

```
class Matrix
{
public:

    Matrix();
    Matrix(int in_rows, int in_cols);
    set_size(int in_rows, int in_cols);

    Matrix(const Matrix& X);           // copy constructor
    const Matrix& operator=(const Matrix& X); // copy operator

    ...

    int    rows;
    int    cols;
    double* data;
};
```

- Overload the `+` operator so we can add two matrices:

```
Matrix operator+(const Matrix& A, const Matrix& B)
{
    // ... check if A and B have the same size ...

    Matrix X(A.rows, A.cols);

    for(int i=0; i < A.rows * A.cols; ++i)
    {
        X.data[i] = A.data[i] + B.data[i];
    }

    return X;
}
```

- Now we can write:

```
Matrix X = A + B;
```

- Done!?! What do we do with the rest of the afternoon?
- Not quite. This approach is actually pretty slow!

- Problem 1: consider what happens here:

```
Matrix X;
```

```
... // do something in the meantime
```

```
X = A + B;
```

- $A + B$ creates a temporary matrix T
 - we've roughly used **twice as much memory** as an optimal (hand coded) solution
- T is then copied into X through the copy operator
 - we've roughly spent **twice as much time** as an optimal solution

- Problem 2: things get worse

```
Matrix X;  
  
... // do something in the meantime  
  
X = A + B + C; // add 3 matrices
```

- $A + B$ creates a temporary matrix $TMP1$
- $TMP1 + C$ creates a temporary matrix $TMP2$
- $TMP2$ is then copied into X through the copy operator
- Obviously we used more memory and more time than really necessary
- How do we solve this ?
 - code algorithms in unreadable low-level C
 - OR: keep readability, use template meta-programming

- First, we need to define a class which does nothing more than hold references to two *Matrix* objects:

```
class Glue
{
public:

    const Matrix& A;
    const Matrix& B;

    Glue(const Matrix& in_A, const Matrix& in_B)
        : A(in_A)
          , B(in_B)
        {
        }

};
```

- Next, we modify the $+$ operator so that instead of producing a matrix, it produces a *const Glue* instance:

```
const Glue operator+(const Matrix& A, const Matrix& B)
{
    return Glue(A,B);
}
```

- Lastly, we modify our matrix class to accept the *Glue* class for construction and copying:

```
class Matrix
{
public:

Matrix();
Matrix(int in_rows, int in_cols);
set_size(int in_rows, int in_cols);

Matrix(const Matrix& X);           // copy constructor
const Matrix& operator=(const Matrix& X); // copy operator

Matrix(const Glue& X);           // copy constructor
const Matrix& operator=(const Glue& X); // copy operator

...

int    rows;
int    cols;
double* data;
};
```

- The additional copy constructor and copy operator will look something like this:

```
// copy constructor
Matrix::Matrix(const Glue& X)
{
    operator=(X);
}

// copy operator
const Matrix&
Matrix::operator=(const Glue& X)
{
    const Matrix& A = X.A;
    const Matrix& B = X.B;

    // ... check if A and B have the same size ...

    set_size(A.rows, A.cols);

    for(int i=0; i < A.rows * A.cols; ++i)
    {
        data[i] = A.data[i] + B.data[i];
    }

    return *this;
}
```

- The *Glue* class holds only *const* references and **operator+** returns a *const Glue*
- The C++ compiler can legally remove temporary and purely **const** instances as long as the results are the same
- By looking at the resulting machine code, it's as if the instance of the *Glue* class **never existed !**
- Hence we can do

```
Matrix X;  
... // do something in the meantime  
X = A + B;
```

without generating temporaries

- What about Problem 2 ?

```
Matrix X;  
... // do something in the meantime  
X = A + B + C; // add 3 matrices
```

- We need to modify the *Glue* class to hold references to two **arbitrary** objects:

```
template <typename T1, typename T2>
class Glue
{
public:

    const T1& A;
    const T2& B;

    Glue(const T1& in_A, const T2& in_B)
        : A(in_A)
          , B(in_B)
        {
        }

};
```

- Note that the class **type** is no longer just **Glue**. It is now **Glue<T1, T2>**

- Next, we modify the $+$ operator to handle the modified *Glue* class:

```
inline
const Glue<Matrix,Matrix>
operator+(const Matrix& A, const Matrix& B)
{
    return Glue<Matrix,Matrix>(A,B);
}
```

- We need to overload the $+$ operator further so we can add a *Glue* object and a *Matrix* object together:

```
inline
const Glue< Glue<Matrix,Matrix>, Matrix>
operator+(const Glue<Matrix,Matrix>& P, const Matrix& Q)
{
    return Glue< Glue<Matrix,Matrix>, Matrix>(P,Q);
}
```

- The result **type** of the expression “A + B” is `Glue<Matrix, Matrix>`
- By doing “A + B + C” we're in effect doing

`Glue<Matrix, Matrix> + Matrix`

which results in a temporary *Glue* instance of **type**:

`Glue< Glue<Matrix, Matrix>, Matrix>`

- We could overload the `+` operator further, allowing for recursive types such as

`Glue< Glue< Glue<Matrix, Matrix>, Matrix>, Matrix >`

- More on this later...

- Our matrix class needs to be modified again

```
class Matrix
{
public:

Matrix();
Matrix(int in_rows, int in_cols);
set_size(int in_rows, int in_cols);

Matrix(const Matrix& X);           // copy constructor
const Matrix& operator=(const Matrix& X); // copy operator

Matrix(const Glue<Matrix,Matrix>& X);
const Matrix& operator=(const Glue<Matrix,Matrix>& X);

Matrix(const Glue< Glue<Matrix,Matrix>, Matrix>& X);
const Matrix& operator=(const Glue< Glue<Matrix,Matrix>, Matrix>& X);

...

int    rows;
int    cols;
double* data;
};
```

- The additional copy constructor and copy operator will look something like this:

```
// copy constructor
Matrix::Matrix(const Glue< Glue<Matrix,Matrix>, Matrix>& X)
{
    operator=(X);
}

// copy operator
const Matrix&
Matrix::operator=(const Glue< Glue<Matrix,Matrix>, Matrix>& X)
{
    const Matrix& A = X.A.A; // first argument of first Glue
    const Matrix& B = X.A.B; // second argument of first Glue
    const Matrix& C = X.B;   // second argument of second Glue

    // ... check if A, B and C have the same size ...

    set_size(A.rows, A.cols);

    for(int i=0; i < A.rows * A.cols; ++i)
    {
        data[i] = A.data[i] + B.data[i] + C.data[i];
    }

    return *this;
}
```

- Okay, so we can do

```
Matrix X;  
... // do something in the meantime  
X = A + B + C;
```

without generating temporaries

- But isn't this approach rather cumbersome ?
(we can't keep extending our *Matrix* class forever)
- What if we want a more general approach ?
(e.g. add 4 matrices, etc)



WARNING:

**THIS WILL GIVE YOU
A HEADACHE**

- We need a way to overload the $+$ operator for all possible combinations of *Glue* and *Matrix*
- The $+$ operator needs to accept arbitrarily long *Glue* types, e.g.:
`Glue< Glue< Glue<Matrix, Matrix>, Matrix>, Matrix>`
- We also need the *Matrix* class to accept arbitrarily long *Glue* types
- First, let's create a strange looking *Base* class:

```
template <typename derived>
struct Base
{
    const derived& get_ref() const
    {
        return static_cast<const derived&>(*this);
    }
};
```

- Second, let's derive the *Matrix* class from the *Base* class:

```
class Matrix : public Base< Matrix >    // for static polymorphism
{
public:

    Matrix();
    Matrix(int in_rows, int in_cols);
    set_size(int in_rows, int in_cols);

    Matrix(const Matrix& X);           // copy constructor
    const Matrix& operator=(const Matrix& X); // copy operator

    ...
    int    rows;
    int    cols;
    double* data;
};
```

- This type of derivation is known as *static polymorphism* (polymorphism without run-time virtual-table lookups)

→ *Matrix M* can be interpreted as *Base<Matrix> M*

- Third, let's derive the *Glue* class from the *Base* class:

```
template <typename T1, typename T2>
class Glue : public Base< Glue<T1, T2 > > // for static polymorphism
{
    public:

    const T1& A;
    const T2& B;

    Glue(const T1& in_A, const T2& in_B)
        : A(in_A)
          , B(in_B)
        {
        }
};
```

- *Glue* $G<Matrix, Matrix>$ can be interpreted as $Base< Glue<Matrix, Matrix> > G$

- We can now define a deceptively simple looking `+` operator:

```
template <typename T1, typename T2>
inline
const Glue<T1, T2>
operator+ (const Base<T1>& A, const Base<T2>& B)
{
    return Glue<T1, T2>( A.get_ref(), B.get_ref() );
}
```

- It can handle arbitrarily long expressions, e.g.:

```
X = A + B + C + D + E + F + G + H + I + J + K + L;
```

- Both the *Glue* and *Matrix* classes are derived from the *Base* class, hence `operator+()` only accepts these two classes
- For variable **A**, which has an arbitrary type `Base<T1>`, calling `A.get_ref()` will give a reference to a variable of type `T1`

- Say we want to add two matrices, i.e.:

```
Matrix A;  
Matrix B;
```

```
Matrix X = A + B;
```

- As **A** can be interpreted as both a *Matrix* and a *Base*, **operator+()** sees **A** as having the type `Base<Matrix>`
- Taking template expansion into account, we're in effect calling **operator+()** as follows:

```
const Glue<Matrix, Matrix>  
operator+ (const Base<Matrix>& A, const Base<Matrix>& B)  
{  
    return Glue<Matrix, Matrix>( A.get_ref(), B.get_ref() );  
}
```

- Inside **operator+()**, calling **A.get_ref()** gives reference to the derived type of `Base<Matrix>`, which is `Matrix`

- Say we want to add three matrices, i.e.:

```
Matrix A;  
Matrix B;  
Matrix C;
```

```
Matrix X = A + B + C;
```

- For the first +, we're in effect calling **operator+()** as:

```
operator+(const Base<Matrix>& A, const Base<Matrix>& B)
```

- produces a temporary of type `Glue<Matrix,Matrix>`

- For the second +, we're in effect calling **operator+()** as:

```
operator+(const Base< Glue<Matrix,Matrix> >& A, const Base<Matrix>& B)
```

- produces a temporary of type `Glue< Glue<Matrix,Matrix>, Matrix>`

- We still need to modify the Matrix class to accept arbitrarily long *Glue* types

```
Glue< Glue< Glue<Matrix, Matrix>, Matrix>, Matrix>
```

- To do that, we first need a way of getting:
 - (a) the number of matrix instances in a *Glue* type
 - (b) the address of each matrix in a *Glue* instance
- For (a), let's adapt the factorial trick we did earlier:

```
template <typename typename T1>
struct depth_lhs
{
    static const int num = 0;           // terminating condition
};
```

```
template <typename T1, typename T2>
struct depth_lhs< Glue<T1,T2> >
{
    // try to expand the left node (T1) which might be a Glue type
    static const int num = 1 + depth_lhs<T1>::num;
};
```

Glue< Glue< Glue<Matrix, Matrix>, Matrix>, Matrix>

- **(b)** the address of each matrix in a *Glue* instance:

```
template <typename T1>
struct mat_ptrs
{
    static const int num = 0;

    inline static void
    get_ptrs(const Matrix** ptrs, const T1& X)
    {
        ptrs[0] = reinterpret_cast<const Matrix*>(&X);
    }
};

template <typename T1, typename T2>
struct mat_ptrs< Glue<T1,T2> >
{
    static const int num = 1 + mat_ptrs<T1>::num;

    inline static void
    get_ptrs(const Matrix** in_ptrs, const Glue<T1,T2>& X)
    {
        // traverse the left node
        mat_ptrs<T1>::get_ptrs(in_ptrs, X.A);

        // get address of the matrix on the right node
        in_ptrs[num] = reinterpret_cast<const Matrix*>(&X.B);
    }
};
```

- Modify our matrix class to accept arbitrarily long *Glue* types:

```
class Matrix : public Base< Matrix >    // for static polymorphism
{
public:

    Matrix();
    Matrix(int in_rows, int in_cols);
    set_size(int in_rows, int in_cols);

    Matrix(const Matrix& X);           // copy constructor
    const Matrix& operator=(const Matrix& X); // copy operator

    template<typename T1, typename T2>
    Matrix(const Glue<T1,T2>& X);

    template<typename T1, typename T2>
    const Matrix& operator=(const Glue<T1,T2>& X);

    ...

    int    rows;
    int    cols;
    double* data;
};
```

- The new copy operator will look something like this:

```
template<typename T1, typename T2>
const Matrix&
Matrix::operator=(const Glue<T1,T2>& X)
{
    int N = 1 + depth_lhs< Glue<T1,T2> >::num;

    const Matrix* ptrs[N];

    mat_ptrs< Glue<T1,T2> >::get_ptrs(ptrs, X);

    int r = ptrs[0]->rows;
    int c = ptrs[0]->cols;

    // ... check that all matrices have the same size ...

    set_size(r, c);

    for(int j=0; j<r*c; ++j)
    {
        double sum = ptrs[0]->data[j];

        for(int i=1; i<N; ++i)
        {
            sum += ptrs[i]->data[j];
        }

        data[j] = sum;
    }

    return *this;
}
```

- That was the tip of the iceberg
- It's also possible to efficiently handle more elaborate matrix expressions
- At NICTA we've made a C++ linear algebra (matrix) library known as **Armadillo**
 - handles ints, floats, doubles and complex numbers
 - interfaces with LAPACK (matrix inversion, etc)
 - about 30,000 lines of code
 - open source
 - available from: <http://arma.sourceforge.net>
<http://arma.sf.net>

- **Lessons learned through developing Armadillo:**
 - It takes a few months to get your head around template meta-programming
 - Heavily templated C++ code can be hard to debug
 - Heavily templated C++ **library** code has little resemblance to C or traditional Java
 - User code (code that uses template libraries) can be much more readable than C or Java
 - **especially scientific code: resembles Matlab !**
 - **less bugs made in user code**
 - Compiling heavily templated code takes typically considerably longer than non-template code
 - Execution speed (run-time) can be ridiculously fast (we've observed speed-ups between 2x to 1000x)
 - Not all C++ compilers can handle heavy duty template meta-programming (e.g. MS Visual Studio has problems)