C++ templates were originally designed to reduce duplication of code

instead of making functions for each type e.g. \textit{float} and \textit{double}

```cpp
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 );
}
```
we can define a function template to 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 );
}

so we've saved ourselves repeating code -> less bugs!

but! the template actually allows more than just float and double...

you can feed it a wrong type by accident -> more bugs!

we will come back to this
• templates can also be used to implement programs that are run at *compile time*

• why would you ever want to do that ??

• example: compute the factorial function, noted as “n!”

• product of a positive integer multiplied by all lesser positive integers, eg. 4! = 4 * 3 * 2 * 1

• traditional implementation:

```c
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
}
```
• **template based** meta-program for computing the factorial:

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

template <> // template specialisation
struct Factorial<0> // 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*
  • but we know 4 at *compile time* -> wasted run time!

• template meta-program:
  • compute factorial at *compile time*
  • smaller code
  • faster execution -> *no wasted run time*!
• we can also use meta-programs to restrict the input types to template functions

```cpp
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);
}
```

• we only want `float` or `double`

• can use SFINAE: substitution failure is not an error

```cpp
template <typename T> struct restrictor { };
template <> struct restrictor<float> { typedef float result; };
template <> struct restrictor<double> { typedef double result; };
```

```cpp
template <typename T>
type name restrictor<T>::result
distance(T a1, T a2, T b1, T b2)
{
    T tmp1 = a1 - b1;
    T tmp2 = a2 - b2;
    return std::sqrt(tmp1*tmp1 + tmp2*tmp2);
}
```
• so how useful is template meta-programming 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:

```cpp
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:

```cpp
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 C++ code that resembles Matlab:

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

• it works... but it has a lot of performance problems!
• problem 1: consider what happens here:

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

• \(A + B\) creates a temporary matrix \(T\)

• \(T\) is then copied into \(X\) through the copy operator

• we've roughly used **twice as much memory** as an optimal (hand coded) solution!

• 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 holds references to two \textit{Matrix} objects:

\begin{verbatim}
class Glue
{
    public:

    const Matrix& A;
    const Matrix& B;

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

};
\end{verbatim}

• Next, we modify the \texttt{+} operator so that instead of producing a matrix, it produces a \textit{const Glue} instance:

\begin{verbatim}
const Glue operator+(const Matrix& A, const Matrix& B)
{
    return Glue(A,B);
}
\end{verbatim}
• Lastly, we modify our matrix class to accept the *Glue* class for construction and copying:

```cpp
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:

```cpp
// 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

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

without generating temporaries -> **problem 1** solved!

• what about **problem 2**?

```cpp
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, instead of two matrices:

```cpp
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 plain *Glue*

• it is now *Glue<T1, T2>*
next, we modify the + operator to handle the modified Glue class:

```cpp
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:

```cpp
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 \( \text{Glue<Matrix, Matrix>} \)

• by doing “A + B + C” we're in effect doing

\[ \text{Glue<Matrix, Matrix>} + \text{Matrix} \]

which results in a temporary \( \text{Glue} \) instance of type:

\[ \text{Glue<Glue<Matrix, Matrix>, Matrix>} \]

• we could overload the + operator further, allowing for recursive types such as

\[ \text{Glue<Glue<Glue<Matrix, Matrix>, Matrix>, Matrix>} \]

• 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:

```cpp
// 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

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

without generating temporary matrices -> problem 2 solved!

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)
• 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, eg:
  
  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);
        }
    };

• function Base<T>::get_ref() will give us a reference to T
• this is a form of static polymorphism
• another way of thinking: Base<T> is a wrapper for class T, where class T can be anything!
second, let's derive the Matrix class from the Base class:

```cpp
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;
};
```

• a Matrix object can be interpreted as a Base<Matrix> object

• function Base<Matrix>::get_ref() will give us a reference to our Matrix object
third, let's derive the *Glue* class from the *Base* class:

```cpp
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)
    {
    }
};
```

- a *Glue*T1,T2> object can be interpreted as a *Base*<Glue*T1,T2> object

- function *Base*<Glue*T1,T2> >::get_ref() will give us a reference to our *Glue*T1,T2> object
we can now define a deceptively simple looking + operator:

```cpp
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() );
}
```

both the Glue and Matrix classes are derived from the Base, hence operator+( ) accepts only Glue and Matrix

recall that Glue doesn't care what it holds references to !
• Glue can hold references to other Glue objects

recall that Base<T> can be parameterised with any type, so we can have Base< Glue< Glue<T1, T2>, T3 > >

operator+( ) can now handle arbitrarily long expressions, eg:

```
X = A + B + C + D + E + F + G + H + I + J + K + L;
```
• say we want to add two matrices, ie:

```cpp
Matrix A;
Matrix B;

Matrix X = A + B;
```

• \textbf{A} can be interpreted as both a \textit{Matrix} and a \textit{Base}, hence \texttt{operator+()} sees \texttt{A} as having the type \texttt{Base<Matrix>}

• taking template expansion into account, we're in effect calling \texttt{operator+()} as follows:

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

• inside \texttt{operator+()}, calling \texttt{A.get_ref()} gives reference to the derived type of \texttt{Base<Matrix>}, which is \texttt{Matrix}
• say we want to add three matrices, ie:

    Matrix A;
    Matrix B;
    Matrix C;

    Matrix X = A + B + C;

• for the first +, we're in effect calling \texttt{operator+()} as:

    \texttt{operator+}(const \texttt{Base<Matrix>>& A, const \texttt{Base<Matrix>>& B)

• produces a temporary of type \texttt{Glue<Matrix,Matrix>}

• for the second +, we're in effect calling \texttt{operator+()} as:

    \texttt{operator+}(const \texttt{Base<Glue<Matrix,Matrix>>}& A, const \texttt{Base<Matrix>>& B)

• produces a temporary of type \texttt{Glue<Glue<Matrix,Matrix>, Matrix>
we still need to modify the Matrix class to accept arbitrarily long \textit{Glue} types

\texttt{Glue< Glue< Glue<Matrix, Matrix>, Matrix>, Matrix>}

to do that, we first need a way of getting:
\begin{itemize}
\item[(a)] the number of matrix instances in a \textit{Glue} type
\item[(b)] the address of each matrix in a \textit{Glue} instance
\end{itemize}

for \textbf{(a)}, let's adapt the factorial meta-program we did earlier:

```cpp
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;
};
```
for (b), the address of each matrix in a Glue instance:

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

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 \textit{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:

```cpp
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 int, float, double and std::complex

  ➔ interfaces with LAPACK (matrix inversion, etc)

  ➔ programs based on Armadillo look like Matlab programs

  ➔ about 85,000 lines of code (125,000 w/ comments, etc)

  ➔ open source (developed w/ contributions from other ppl)

  ➔ available from: http://arma.sourceforge.net
• Lessons learned through developing Armadillo:

• it takes a few months to get your head around template meta-programming

• template meta-programming generally requires a higher cognitive load: you need to think about possible template expansions, in addition to normal program logic

• heavily templated C++ library code has little resemblance to C or traditional Java, or the pure OOP subset of C++

  ➔ the number of people that can understand heavy template code is relatively small: possible maintenance issue

• heavily templated C++ code can be hard to debug, if deliberate precautions are not taken!

  ➔ GNU C++ (GCC) compiler comes in very handy: can print out exact function signatures, including all template parameters
• **user code** (code that uses template libraries) is much more readable than C or Java
  → especially scientific/algorithm code: resembles Matlab!
  → faster to write user code
• less bugs in user code

• **compiling** heavy templates takes longer than non-template code
  → C++ compilers are improving: slowness is becoming less of an issue
  → clang is quite fast

• **execution speed** (run-time) of template-based programs can be very fast (we've observed speed-ups between 2x to 1000x)

• not all C++ compilers can properly handle heavy template meta-programming:
  • Borland C++ builder has problems
  • MS Visual C++ has lots of problems (mainly versions prior to 2013)
    → quite sad that a company as big as Microsoft was unable to properly implement a C++ compiler for many, many years! Internal culture problem?

• **recommended compilers:**
  • GCC (Linux, Mac OS X, Windows)
  • clang (Linux, Mac OS X)
  • Intel C++ compiler
• Full source code for the Armadillo template library:
  ➔ http://arma.sourceforge.net

•Related publications:
  ➔ C. Sanderson.
  Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping
  and Computationally Intensive Experiments.

  ➔ D. Eddelbuettel, C. Sanderson.
  RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra.
  http://dx.doi.org/10.1016/j.csda.2013.02.005

• Questions? Comments?
  ➔ contact me: http://conradsanderson.id.au