Template metaprograms

- First template metaprogram: Erwin Unruh 1994
  - Printed prime numbers as compiler error messages
- Later proved to be a Turing-complete sublanguage of C++
- Today many use cases
  - Expression templates
  - DSL (boost::xpressive)
  - Generators (boost::spirit)
  - Compile-time adaptation (std::enable_if)
- Many more...

Factorial

- Compile time recursion
- Specialization to stop recursion

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N-1>::value };
};
template <>
struct Factorial<1>
{
    enum { value = 1 };
};

// example use
int main()
{
    const int fact5 = Factorial<5>::value;
    std::cout << fact5 << endl;
    return 0;
}
```

Meta-control structures

```
template <bool condition, class Then, class Else>
struct IF
{
    typedef Then RET;
};

template <class Then, class Else>
struct IF<false, Then, Else>
{
    typedef Else RET;
};

// example use
template <typename T, typename S>
IF< sizeof(T)<sizeof(S), S, T>::RET max(T t, S s)
{
    if ( t < s)
        return s;
    else
        return t;
}
```

Template metafunction as parameter

```
template <int n, template<int> class F>
struct Accumulate
{
    enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template <template<int> class F>
struct Accumulate<0,F>
{
    enum { RET = F<0>::RET };
};

template <int n>
struct Square
{
    enum { RET = n*n };
};

cout << Accumulate<3,Square>::RET << endl;
```

Why use it?

```
template <unsigned long N>
struct binary
{
    static unsigned const value = binary<N/10>::value * 2 + N % 10;
};

template <>
struct binary<0>
{
    static unsigned const value = 0;
};

int main()
{
    const unsigned int di = 12;
    const unsigned int oi = 014;
    const unsigned int hi = 0xc;

    const unsigned int bi0 = binary_value("1101");  // run-time
    const unsigned int bi1 = binary<1100>::value;   // compile-time
}
```

Use case example

```
template <class T>
class matrix
{
public:
    matrix( int i, int j );
    matrix( const matrix &other);
    ~matrix();
    matrix operator=( const matrix &other);
private:
    int  x;
    int  y;
    T   *v;
    void copy( const matrix &other);
    void check( int i, int j) const throw(indexError);
};
```

Specialization for POD types

```
template <class T>
void matrix<T>::copy( const matrix &other)
{
    x = other.x;
    y = other.y;
    v = new T[x*y];
    for ( int i = 0; i < x*y; ++i )
       v[i] = other.v[i];
}

// specialization for POD types
template <>
void matrix<long>::copy( const matrix &other)
{
    x = other.x;
    y = other.y;
    v = new long[x*y];
    memcpy( v, other.v, sizeof(long)*x*y);
}

template <>
void matrix<int>::copy( const matrix &other) ...
```

Trait

```
template <typename T> struct copy_trait
{
    static void copy( T* to, const T* from, int n)  {
        for( int i = 0; i < n; ++i ) to[i] = from[i];
    }
};
template <> struct copy_trait<long>
{
    static void copy( long* to, const long* from, int n) {
        memcpy( to, from, n*sizeof(long));
    }
};
template <class T, class Cpy = copy_trait<T> >
class matrix {  ... }

template <class T, class Cpy>
void matrix<T,Cpy>::copy( const matrix &other) {
    x = other.x;
    y = other.y;
    v = new T[x*y];
    Cpy::copy( v, other.v, x*y);
}
```

Policy

```
template <typename T, bool B> struct copy_trait
{
    static void copy( T* to, const T* from, int n) {
        for( int i = 0; i < n; ++i )
            to[i] = from[i];
    }
};
template <typename T> struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)    {
        memcpy( to, from, n*sizeof(T));
    }
};

template <typename T> struct is_pod {  enum { value = false }; };
template <> struct is_pod<long>     {  enum { value = true }; };

template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix  { ... }
```

Typelist

```
class NullType {};

// We now can construct a null-terminated list of typenames:
typedef Typelist< char,
           Typelist<signed char,
             Typelist<unsigned char, NullType>
           >
        > Charlist;

// For the easy maintenance, precompiler macros are defined
// to create Typelists:

#define TYPELIST_1(x)           Typelist< x, NullType>
#define TYPELIST_2(x, y)        Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)     Typelist< x, TYPELIST_2(y,z)>
#define TYPELIST_4(x, y, z, w)  Typelist< x, TYPELIST_3(y,z,w)>

// usage example
typedef TYPELIST_3(char, signed char, unsigned char)  Charlist;
```

Typelist operations

```
//  Length
template <class TList> struct Length;

template <>
struct Length<NullType>
{
    enum { value = 0 };
};

template <class T, class U>
struct Length <Typelist<T,U> >
{
    enum { value = 1 + Length<U>::value };
};

static const int len = Length<Charlist>::value;
```

Typelist operations

```
// IndexOf
template <class TList, class T> struct IndexOf;

template <class T>
struct IndexOf< NullType, T>
{
    enum { value = -1 };
};
template <class T>
struct IndexOf< Typelist<T, Tail>, T>
{
    enum { value = 0 };
};
template <class T, class Tail>
struct IndexOf< Typelist<Head, Tail>, T>
{
private:
    enum { temp = IndexOf<Tail, T>::value };
public:
    enum { value = (temp == -1 ? -1 : 1+temp) };
};

static const int IndexOf<Charlist, int>::value;            // -1
static const int IndexOf<Charlist, char>::value;           //  0
static const int IndexOf<Charlist, unsigned char>::value;  //  2
```

Matrix revisited

```
typedef TYPELIST_4(char, signed char, unsigned char, int)  Pod_types;

template <typename T> struct is_pod
{
    enum { value = IndexOf<Pod_types,T>::value != -1 };
};
template <typename T, bool B> struct copy_trait
{
    static void copy( T* to, const T* from, int n)   {
        for( int i = 0; i < n; ++i )
            to[i] = from[i];
    }
};
template <typename T> struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)   {
        memcpy( to, from, n*sizeof(T));
    }
};
template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix { ... }
```