

Template Metaprogramming in C++

CS242, Fall 2009
Keith Schwarz

Preliminaries

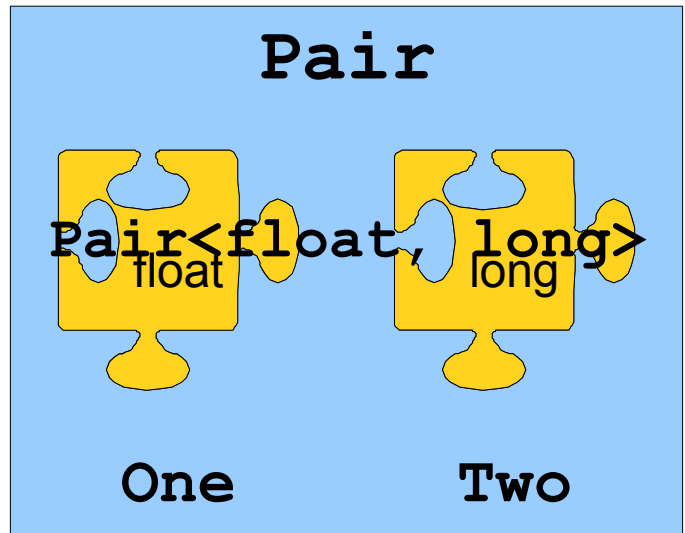
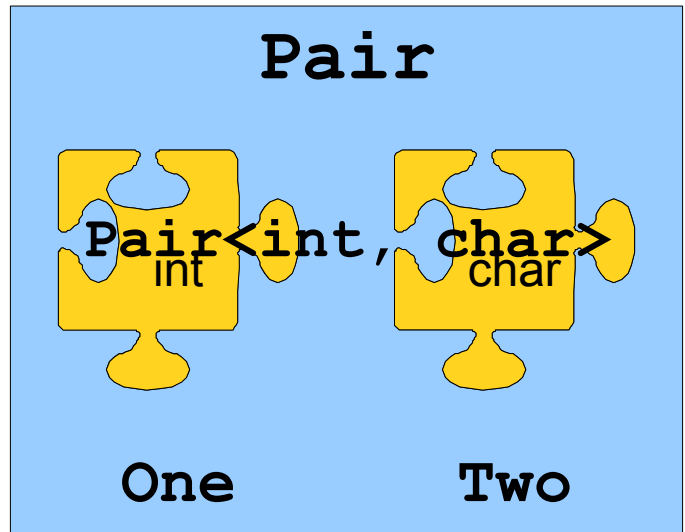
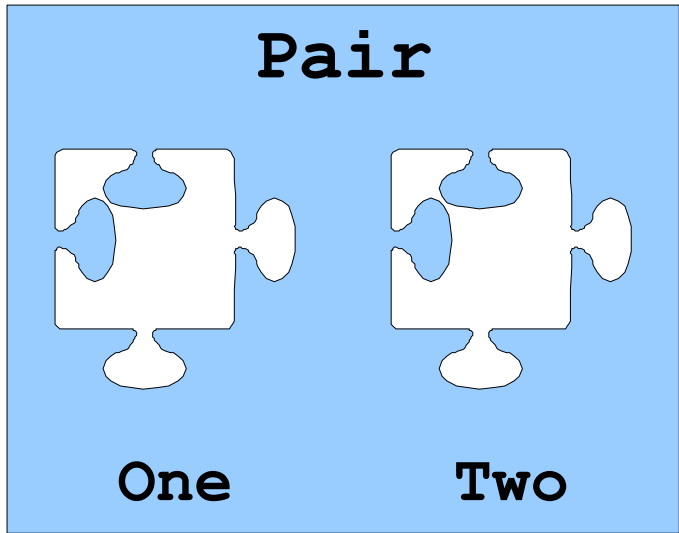
A **C++ template** is a type or function parameterized over a set of types, functions, or constants.

```
template <typename One, typename Two>
struct Pair
{
    One first;
    Two second;
};
```

```
template <typename One, typename Two>  
struct Pair  
{  
    One first;  
    Two second;  
};
```

```
template <typename One, typename Two>
struct Pair
{
    One first;
    Two second;
};
```

Providing arguments to a template **instantiates** the template with those arguments. Instantiation occurs at **compile-time**.



Template Specialization

- A version of a template to use when a specific pattern of arguments are supplied.
- Structure independent of primary template.
 - Can add/remove functions from interface, etc.
- *Full specialization* used when all arguments are specified.
- *Partial specialization* used when arguments have a particular structure.

```
/* Primary template */  
template <typename T> class Set  
{  
    // Use a binary tree  
};
```

```
/* Primary template */  
template <typename T> class Set  
{  
    // Use a binary tree  
};
```

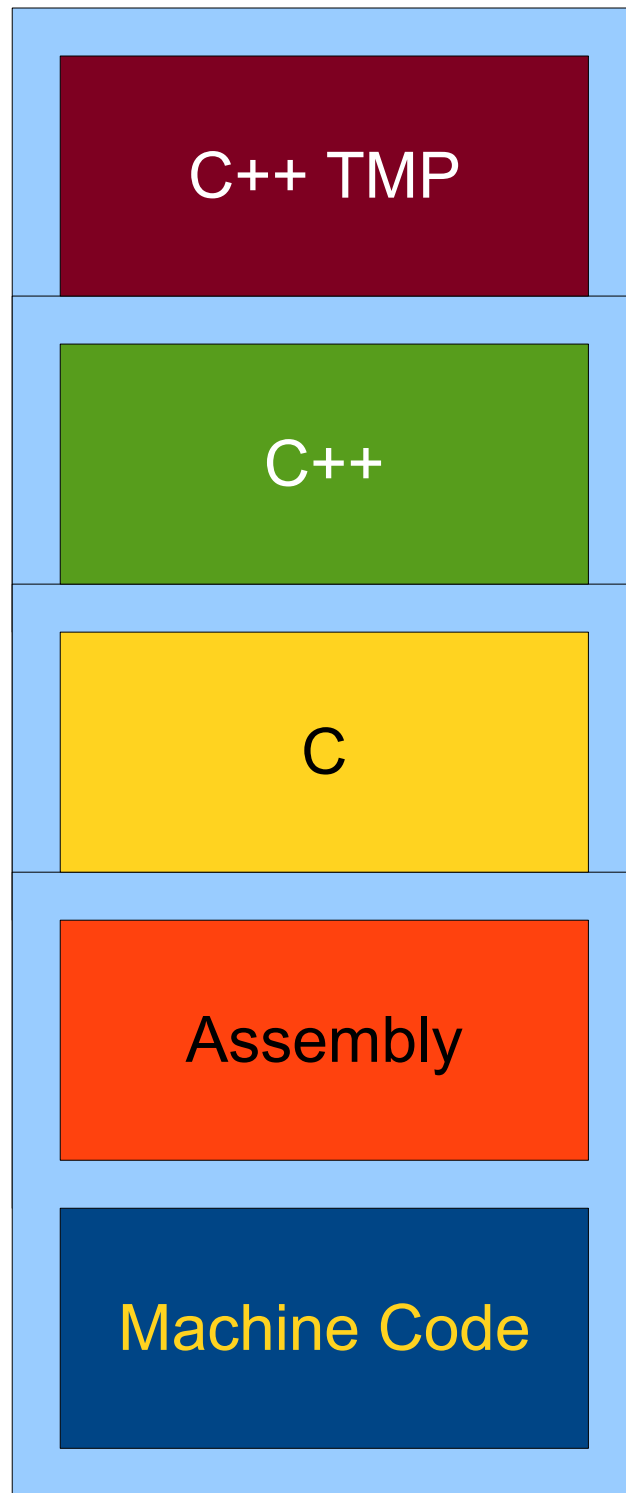
```
/* Full specialization */  
template <> class Set<char>  
{  
    // Use a bit vector  
};
```

```
/* Primary template */  
template <typename T> class Set  
{  
    // Use a binary tree  
};  
  
/* Full specialization */  
template <> class Set<char>  
{  
    // Use a bit vector  
};  
  
/* Partial specialization */  
template <typename T> class Set<T*>  
{  
    // Use a hash table  
};
```

A **metaprogram** is a program that produces or manipulates constructs of a target language.

A **template metaprogram** is a C++ program that uses templates to generate customized C++ code at compile-time.

Why would you ever want to do this?



Template Metaprogramming In Action

Part One: Policy Classes

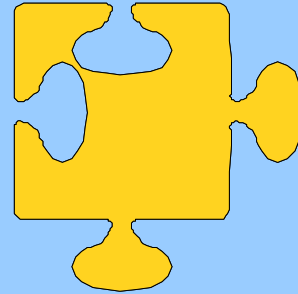
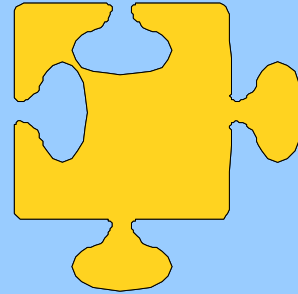
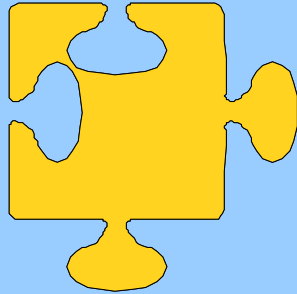
```
template <typename T> class Vector
{
public:
    /* ... ctors, dtor, etc. */
        T& operator[] (size_t);
    const T& operator[] (size_t) const;

    void insert(iterator where,
                const T& what);

    /* ... etc. ... */
};
```

Vector

Type T



Range Checking

Synchronization

Templates are parameterized over **types**, not **behaviors**.

A **policy class** is a type that implements a particular behavior.

```
template <typename T>
class Vector

{
public:
    /* ... ctors, dtor, etc. */
        T& operator[] (size_t);
    const T& operator[] (size_t) const;

    void insert(iterator where,
                const T& what);

    /* ... etc. ... */
};
```

```
template <typename T,  
          typename RangePolicy,  
          typename LockingPolicy>  
class Vector  
  
{  
public:  
    /* ... ctors, dtor, etc. */  
    T& operator[] (size_t);  
    const T& operator[] (size_t) const;  
  
    void insert(iterator where,  
                const T& what);  
  
    /* ... etc. ... */  
};
```

```
template <typename T,  
          typename RangePolicy,  
          typename LockingPolicy>  
class Vector: public RangePolicy,  
              public LockingPolicy  
{  
public:  
    /* ... ctors, dtor, etc. */  
        T& operator[] (size_t);  
    const T& operator[] (size_t) const;  
  
    void insert(iterator where,  
                const T& what);  
  
    /* ... etc. ... */  
};
```

Sample Range Policy

```
class ThrowingErrorPolicy
{
protected:
    ~ThrowingErrorPolicy() {}

    static void CheckRange(size_t pos,
                           size_t numElems)
    {
        if (pos >= numElems)
            throw std::out_of_bounds("Bad!");
    }
};
```

Another Sample Range Policy

```
class LoggingErrorPolicy
{
public:
    void setLogFile(const std::string&);
protected:
    ~LoggingErrorPolicy();
    void CheckRange(size_t pos,
                    size_t numElems)
    {
        if (pos >= numElems && output != 0)
            *log << "Error!" << std::endl;
    }
private:
    std::ofstream* log;
};
```

Another Sample Range Policy

```
class LoggingErrorPolicy
{
public:
    void setLogFile(const std::string&);
protected:
    ~LoggingErrorPolicy();
    void CheckRange(size_t pos,
                    size_t numElems)
    {
        if (pos >= numElems && output != 0)
            *log << "Error!" << std::endl;
    }
private:
    std::ofstream* log;
};
```

Implementer Code

```
template <typename T,  
         typename RangePolicy,  
         typename LockingPolicy>  
T& Vector<T, RangePolicy, LockingPolicy>::  
    operator[] (size_t position)  
{  
  
    return this->elems[position];  
}
```

Implementer Code

```
template <typename T,  
         typename RangePolicy,  
         typename LockingPolicy>  
T& Vector<T, RangePolicy, LockingPolicy>::  
    operator[] (size_t position)  
{  
    LockingPolicy::Lock lock;  
  
    return this->elems[position];  
}
```

Implementer Code

```
template <typename T,  
         typename RangePolicy,  
         typename LockingPolicy>  
T& Vector<T, RangePolicy, LockingPolicy>::  
    operator[] (size_t position)  
{  
    LockingPolicy::Lock lock;  
    RangePolicy::CheckRange(position,  
                            this->size);  
    return this->elems[position];  
}
```

Client Code

```
int main()
{
    Vector<int, ThrowingErrorPolicy,
        NoLockingPolicy> v;

    for(size_t k = 0; k < kNumElems; ++k)
        v.push_back(k);

    /* ... etc. ... */

    return 0;
}
```

```
template <
    typename T,
    typename RangePolicy = NoErrorPolicy,
    typename LockingPolicy = NoLockingPolicy>
class Vector: public RangePolicy,
              public LockingPolicy
{
public:
    /* ... ctors, dtor, etc. */
        T& operator[] (size_t);
    const T& operator[] (size_t) const;

    void insert(iterator where,
                const T& what);

    void erase(iterator where);

    /* ... etc. ... */
};
```

Updated Client Code

```
int main()
{
    Vector<int, ThrowingErrorPolicy> v;

    for(size_t k = 0; k < kNumElems; ++k)
        v.push_back(k);

    /* ... etc. ... */

    return 0;
}
```

Summary of Policy Classes

- Identify **mutually orthogonal** behaviors in a class.
- Specify an **implicit interface** for those behaviors.
- **Parameterize** a host class over each policy.
- Use **multiple inheritance** to import the policies into the host.

Template Metaprogramming In Action

Part Two: Traits Classes and Tag Dispatching

```
template </* ... */>
class Vector: /* ... */
{
public:
    void insert(iterator where,
                const T& what);
```

```
template <typename IteratorType>
void insert(iterator where,
            IteratorType start,
            IteratorType stop);
```

```
/* ... */
};
```

```
template <...>
    template <typename Iter>
void Vector<...>::insert(iterator where,
                        Iter start,
                        Iter stop)
{
    /* Insert elements one at a time. */
    for(; start != stop; ++start, ++where)
        where = insert(where, start);
}
```

```
template <...>
    template <typename Iter>
void Vector<...>::insert(iterator where,
                        Iter start,
                        Iter stop)
{
    /* Insert elements one at a time. */
    for(; start != stop; ++start, ++where)
        where = insert(where, start);
}
```

```
template <typename Iter> struct iterator_traits
{
    typedef typename Iter::difference_type
        difference_type;
    typedef typename Iter::value_type value_type;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::reference reference;
    typedef typename Iter::iterator_category
        iterator_category;
};

/* Specialization for raw pointers */
template <typename T> struct iterator_traits<T*>
{
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef random_access_iterator_tag
        iterator_category;
};
```

A **traits class** is a template type that exports information about its parameters.

Schematic of Traits Classes



```
template <typename Iter> struct iterator_traits
{
    typedef typename Iter::difference_type
        difference_type;
    typedef typename Iter::value_type value_type;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::reference reference;
    typedef typename Iter::iterator_category
        iterator_category;
};

/* Specialization for raw pointers */
template <typename T> struct iterator_traits<T*>
{
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef random_access_iterator_tag
        iterator_category;
};
```

```
template <typename Iter> struct iterator_traits
{
    typedef typename Iter::difference_type
        difference_type;
    typedef typename Iter::value_type value_type;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::reference reference;
typedef typename Iter::iterator_category
    iterator_category;
};

/* Specialization for raw pointers */
template <typename T> struct iterator_traits<T*>
{
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
typedef random_access_iterator_tag
    iterator_category;
};
```

```
struct input_iterator_tag {};  
  
struct output_iterator_tag {};  
  
struct forward_iterator_tag :  
    input_iterator_tag, output_iterator_tag {};  
  
struct bidirectional_iterator_tag :  
    forward_iterator_tag {};  
  
struct random_access_iterator_tag :  
    bidirectional_iterator_tag {};
```

A **tag class** is a (usually empty) type encoding semantic information.

Tag dispatching is function overloading on tag classes.

```
template <...>
    template <typename Iter>
void Vector<...>::insert(iterator where,
                        Iter start,
                        Iter stop)
{
    doInsert(where, start, stop,
            typename std::iterator_traits<Iter>::iterator_category());
}
```

```
template <...>
    template <typename Iter>
void Vector<...>::insert(iterator where,
                        Iter start,
                        Iter stop)
{
    doInsert(where, start, stop,
            typename std::iterator_traits<Iter>::iterator_category());
}
```

```

template <...>
    template <typename Iter>
void Vector<...>::doInsert(iterator where,
                          Iter start, Iter stop,
                          std::input_iterator_tag)
{
    /* Insert elements one at a time. */
    for(; start != stop; ++start, ++where)
        where = insert(where, start);
}

template <...>
    template <typename Iter>
void Vector<...>::doInsert(iterator where,
                          Iter start, Iter stop,
                          std::forward_iterator_tag)
{
    /* ... more complex logic to shift everything
     * down at the same time...
     */
}

```

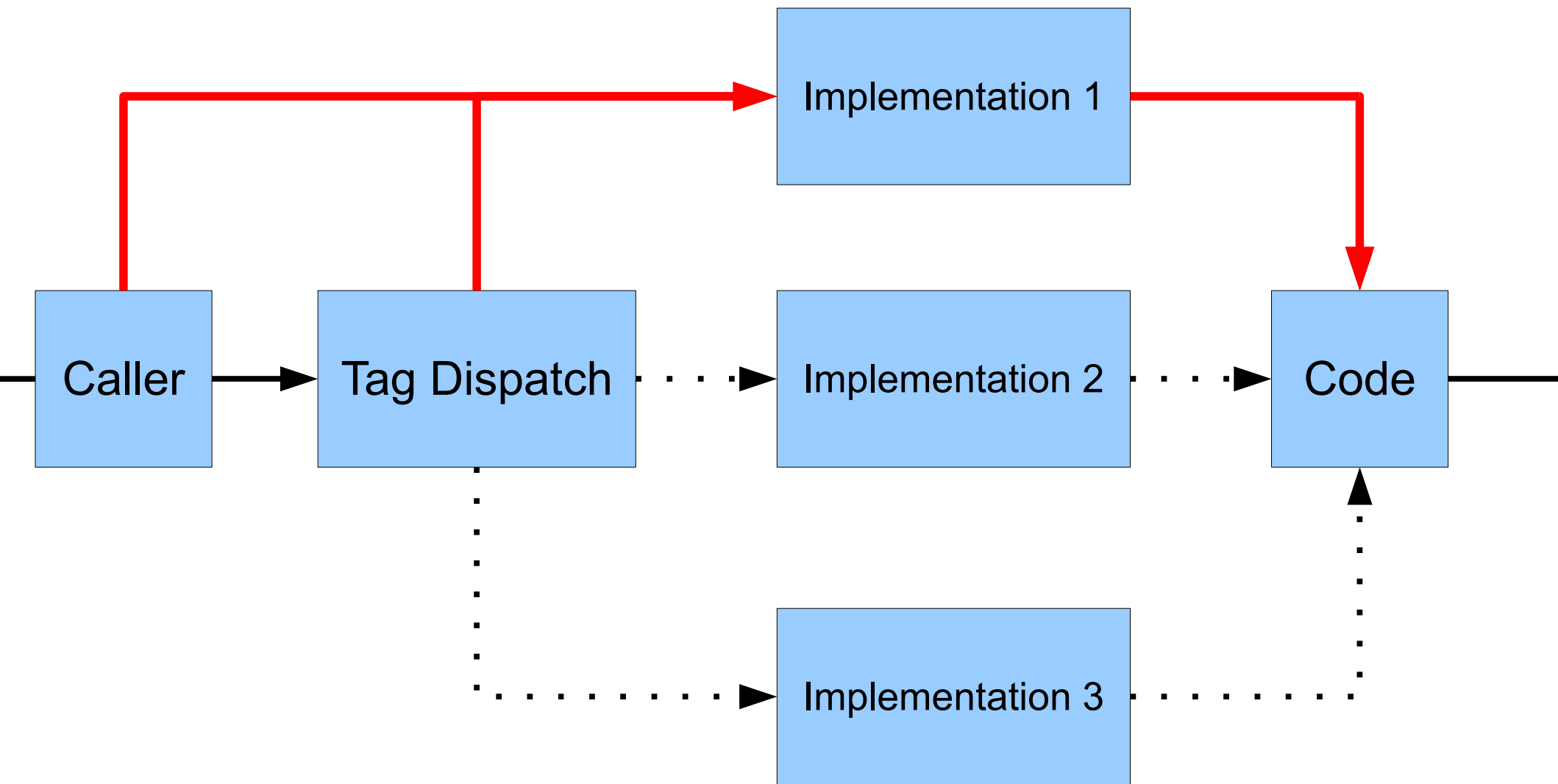
```

template <...>
    template <typename Iter>
void Vector<...>::doInsert(iterator where,
                          Iter start, Iter stop,
                          std::input_iterator_tag)
{
    /* Insert elements one at a time. */
    for(; start != stop; ++start, ++where)
        where = insert(where, start);
}

template <...>
    template <typename Iter>
void Vector<...>::doInsert(iterator where,
                          Iter start, Iter stop,
                          std::forward_iterator_tag)
{
    /* ... more complex logic to shift everything
     * down at the same time...
     */
}

```

Schematic of Tag Dispatching



Summary of Tag Dispatching

- Define a set of tag classes encoding **semantic information**.
- Provide a means for obtaining a tag from each relevant type (often using **traits classes**)
- **Overload the relevant function** by accepting different tag types as parameters.
- Call the overloaded function using the tag associated with each type.

Template Metaprogramming In Action

Part Three: Typelists

The Typelist

```
struct Nil {};  
template <typename Car, typename Cdr>  
    struct Cons {};
```

Sample Typelist

```
Cons<int,  
    Cons<double,  
        Cons<char,  
            Cons<float,  
                Cons<short,  
                    Cons<long, Nil>  
                >  
            >  
        >  
    >  
>
```

A Simplification

```
#define LIST0 () Nil
#define LIST1 (a) Cons<a, LIST0 ()>
#define LIST2 (a, b) Cons<a, LIST1 (b)>
#define LIST3 (a, b, c) Cons<a, LIST2 (b, c)>
#define LIST4 (a, b, c, d) Cons<a, LIST3 (b, c, d)>
/* ... etc. ... */
```

```
LIST6(int, double, float, char, short, long)
```

Car/Cdr Recursion with Templates

```
template <typename> struct Length;
```

Car/Cdr Recursion with Templates

```
template <typename> struct Length;
```

```
template <> struct Length<Nil>  
{  
    static const size_t result = 0;  
};
```

Car/Cdr Recursion with Templates

```
template <typename> struct Length;
```

```
template <> struct Length<Nil>  
{  
    static const size_t result = 0;  
};
```

```
template <typename Car, typename Cdr>  
struct Length<Cons<Car, Cdr> >  
{  
    static const size_t result =  
        1 + Length<Cdr>::result;  
};
```

Length<LIST3(int, double, string)>

result

Length<LIST2(double, string)>

result

Length<LIST1(string)>

result

Length<LIST0()>

result

Length<LIST3(int, double, string)>

result

3

Length<LIST2(double, string)>

result

2

Length<LIST1(string)>

result

1

Length<LIST0()>

result

0

Typelists and template specialization allow us to write templates whose instantiation causes a **chain reaction** of further instantiations.

This lets us construct **arbitrarily complicated structures** at compile-time.

```
class ExprVisitor;

class ExprNode {
public:
    virtual void accept(ExprVisitor&);
};

class AddExpr: public ExprNode {
public:
    virtual void accept(ExprVisitor&);
};

class MulExpr: public ExprNode {
public:
    virtual void accept(ExprVisitor&);
};

class SubExpr: public ExprNode {
public:
    virtual void accept(ExprVisitor&);
};

class DivExpr: public ExprNode {
public:
    virtual void accept(ExprVisitor&);
};
```

```
class ExprVisitor
{
public:
    virtual void visit (ExprNode*) = 0;
    virtual void visit (AddExpr*) = 0;
    virtual void visit (MulExpr*) = 0;
    virtual void visit (SubExpr*) = 0;
    virtual void visit (DivExpr*) = 0;
}
```

```
void ExprNode::accept (ExprVisitor& v)
{
    v.visit(this); // Calls ExprVisitor::Visit (ExprNode*)
}

void AddExpr::accept (ExprVisitor& v)
{
    v.visit(this); // Calls ExprVisitor::Visit (AddExpr*)
}

void MulExpr::accept (ExprVisitor& v)
{
    v.visit(this); // Calls ExprVisitor::Visit (MulExpr*)
}

void DivExpr::accept (ExprVisitor& v)
{
    v.visit(this); // Calls ExprVisitor::Visit (DivExpr*)
}

void SubExpr::accept (ExprVisitor& v)
{
    v.visit(this); // Calls ExprVisitor::Visit (SubExpr*)
}
```

```
class FSVisitor;

class FileSystemEntity {
public:
    virtual void accept (FSVisitor&);
};

class File: public FileSystemEntity {
public:
    virtual void accept (FSVisitor&);
};

class Directory: public FileSystemEntity {
public:
    virtual void accept (FSVisitor&);
};

/* ... etc. ... */
```

```
class FSVisitor
{
public:
    virtual void visit(FileSystemEntity*) = 0;
    virtual void visit(File*) = 0;
    virtual void visit(Directory*) = 0;
    /* ... etc. ... */
}
```

Can we automatically generate a visitor for a type hierarchy?

Yes!

Idea: Create a type **parameterized over a `typelist`** that has one instance of `visit` for each type in the list.

```
template <typename List> class Visitor;
```

```
template <typename List> class Visitor;
```

```
template <typename T> class Visitor<LIST1(T) >  
{  
public:  
    virtual ~Visitor() {}  
    virtual void visit(T*) = 0;  
};
```

```
template <typename List> class Visitor;
```

```
template <typename T> class Visitor<LIST1(T) >  
{  
public:  
    virtual ~Visitor() {}  
    virtual void visit(T*) = 0;  
};
```

```
template <typename Car, typename Cdr>  
class Visitor<Cons<Car, Cdr> > : public Visitor<Cdr>  
{  
public:  
    virtual void visit(Car*) = 0;  
    using Visitor<Cdr>::visit;  
};
```

Visitor<LIST1(ExprNode)>

`virtual void visit(ExprNode*)`

Visitor<LIST2(DivExpr, ExprNode)>

`virtual void visit(DivExpr*)`

Visitor<LIST3(SubExpr, DivExpr, ExprNode)>

`virtual void visit(SubExpr*)`

Visitor<LIST4(MulExpr, SubExpr, DivExpr, ExprNode)>

`virtual void visit(MulExpr*)`

Visitor<LIST5(AddExpr, MulExpr, SubExpr, DivExpr, ExprNode)>

`virtual void visit(AddExpr*)`

**Visitor<LIST5(AddExpr, MulExpr,
SubExpr, DivExpr, ExprNode)>**

```
virtual void visit(AddExpr*)  
virtual void visit(MulExpr*)  
virtual void visit(SubExpr*)  
virtual void visit(DivExpr*)  
virtual void visit(ExprNode*)
```

Summary of Typelists

- Construct types corresponding to **LISP-style lists** whose elements are types.
- Use **template specialization** to model car/cdr recursion.

The Limits of Template Metaprogramming

Queue Automaton

- A **queue automaton** is a finite-state machine equipped with a **queue**.
 - Contrast to PDA with a **stack**.
- Tuple $(Q, \Sigma, \Gamma, \$, q_0, \delta)$ where
 - Q is a set of states
 - Σ is the *input alphabet*
 - Γ is the *tape alphabet*
 - $\$ \in \Gamma - \Sigma$ is the *start symbol*
 - $q_0 \in Q$ is the *start state*
 - $\delta \in Q \times \Gamma \rightarrow Q \times \Gamma^*$ is the *transition function*

Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX

Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: **XXX**



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: **XXX**



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: **XXX**



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: **XXX**



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: **XXX**



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

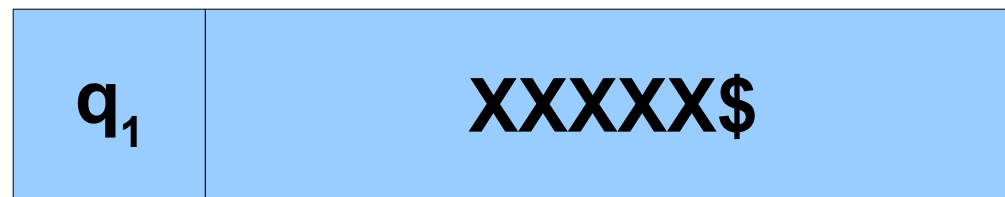
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

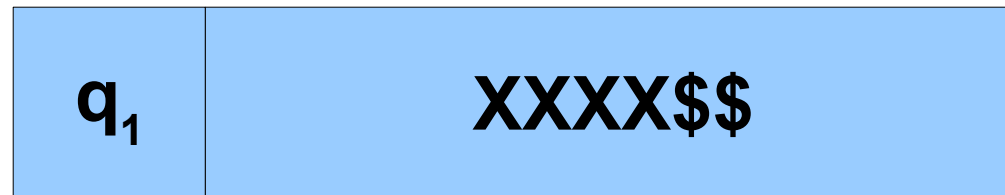
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

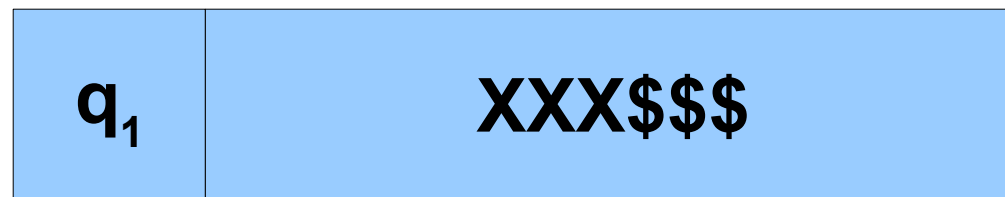
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

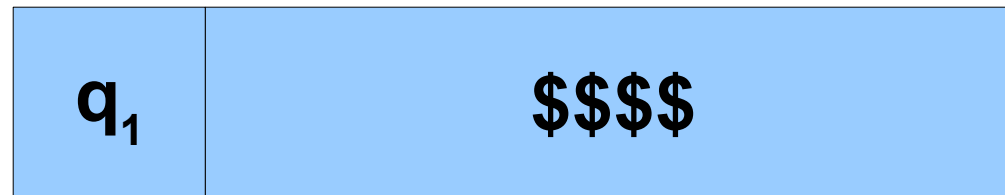
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

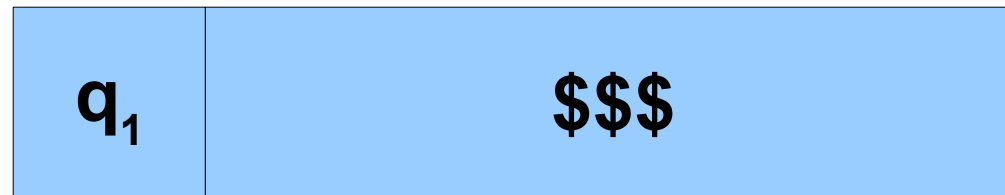
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

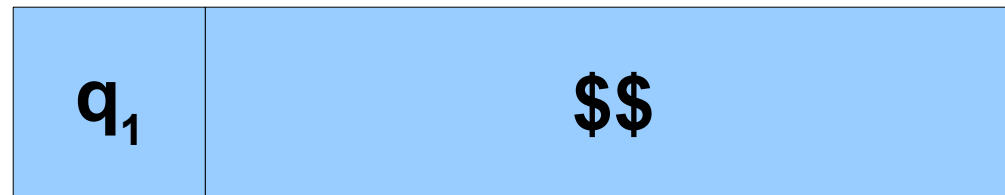
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

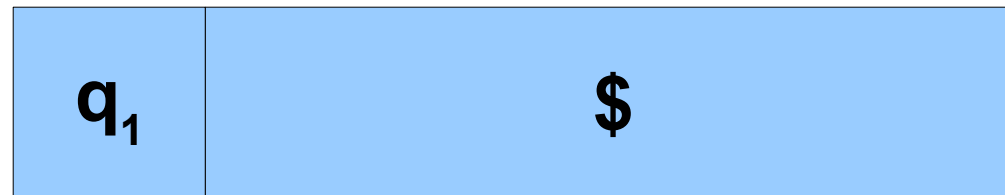
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

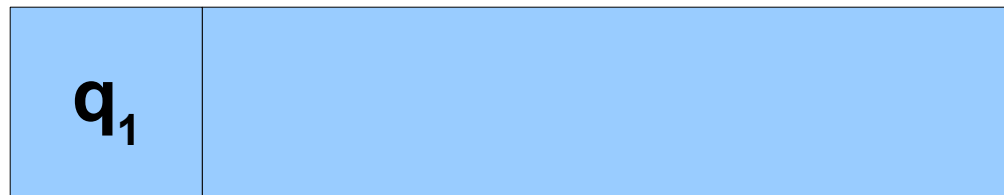
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

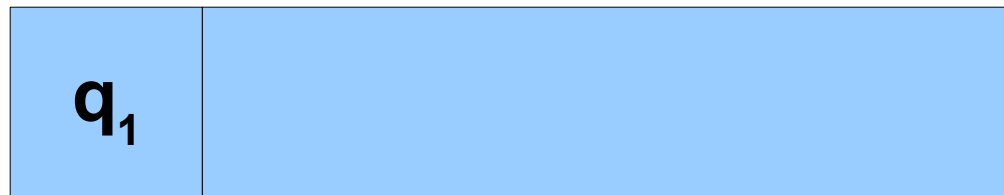
Input: XXX



Queue Automaton Example

δ	X	\$
q_0	q_0, XX	q_1, ϵ
q_1	$q_1, \$$	q_1, ϵ

Input: XXX



ACCEPT

Can we simulate a queue automaton with a
template metaprogram?

Yes!

Concatenating Two Typelists

```
template <typename, typename> struct Concat;
template <typename T> struct Concat<Nil, T>
{
    typedef T result;
};
template <typename Car, typename Cdr, typename T>
struct Concat<Cons<Car, Cdr>, T>
{
    typedef Cons<Car, typename Concat<Cdr, T>::result>
        result;
};
```

Encoding $Q, \Sigma, \Gamma, \$, q_0$

```
/* Define a tag type for each state in  $Q$  */
struct State1 {};
struct State2 {};
    ...
struct StateN {};

/* Define a tag type for each symbol in  $\Sigma \cup \Gamma$  */
struct Symbol1 {};
struct Symbol2 {};
    ...
struct SymbolN {};

/* Designate  $q_0$  and  $\$$ . */
struct StartState {};
struct StartSymbol{};
```

Encoding the Transition Table δ

```
template <typename, typename> struct Delta;  
  
/* Specialize Delta for each entry in  $\delta$  */  
template <> struct Delta<State1, Symbol1>  
{  
    typedef State2 nextState;  
    typedef LIST2(Symbol1, Symbol1) nextSymbols;  
};  
  
template <> struct Delta<State1, Symbol2>  
{  
    typedef State1 nextState;  
    typedef LIST0() nextSymbols;  
};  
  
    /* ... etc. ... */
```

Running the Queue Automaton

```
template <typename State, typename Queue>
struct RunAutomaton;
```

```
template <typename State>
struct RunAutomaton<State, Nil>
{
    typedef void result;
};
```

```
template <typename Car, typename Cdr, typename State>
struct RunAutomaton<State, Cons<Car, Cdr> >
{
    typedef typename Delta<State, Car>::nextState newState;
    typedef typename Delta<State, Car>::nextSymbols newSym;
    typedef typename Concat<Cdr, newSym>::result newQueue;

    typedef
        typename RunAutomaton<newState, newQueue>::result result;
};
```

Starting the Queue Automaton

```
template <typename T>
struct SimulateQueueAutomaton
{
    typedef typename Concat<T, LIST1(StartSymbol)>::result
        initialQueue;

    typedef typename
        RunAutomaton<StartState, initialQueue>::result result;
};
```

A Turing machine can simulate C++ templates.

C++ templates can simulate queue automata.

Queue automata can simulate Turing machines.

C++ templates are Turing-complete.

In other words, the C++ type system has the **same computational capabilities** as C++.

Applications of TMP

- Compile-time dimensional analysis.
- Multiple dispatch.
- Optimized matrix operations.
- Domain-specific parsers.
- Compiler-enforced code annotations.
- Optimized FFT.

Questions?