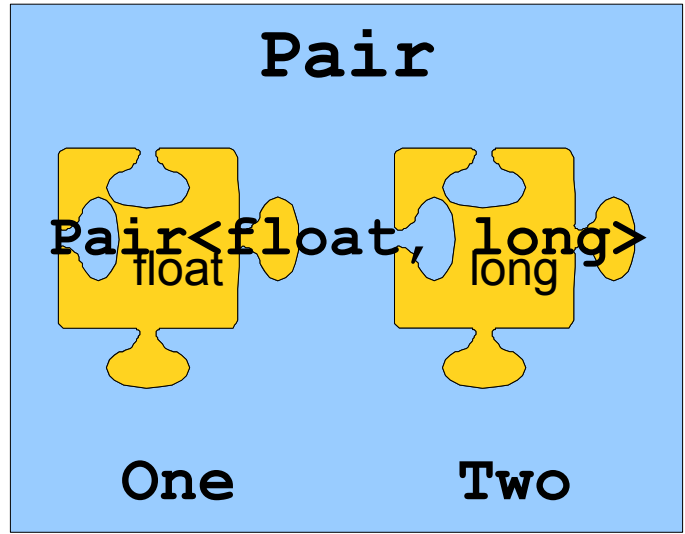
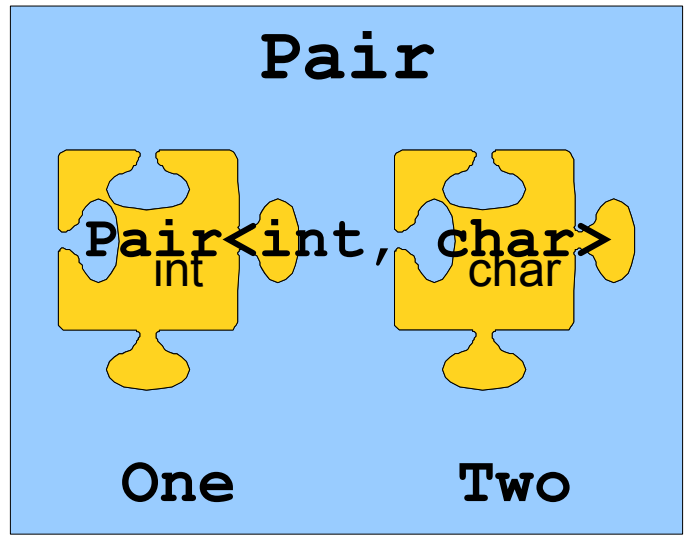
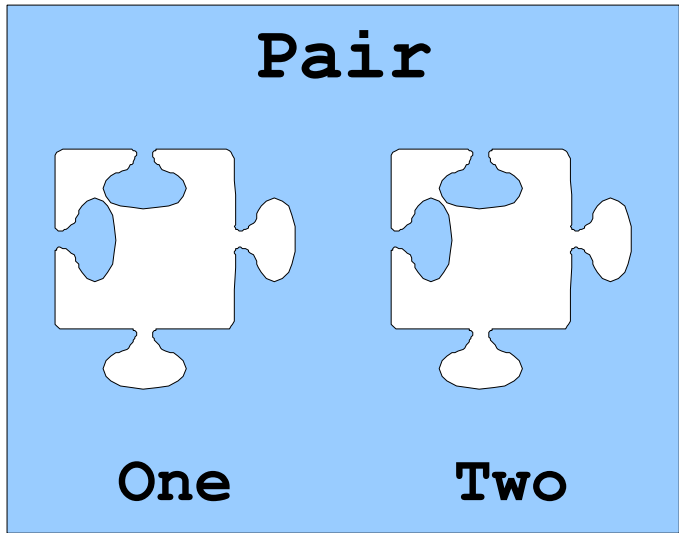


# Template Metaprogramming

# Preliminaries

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

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

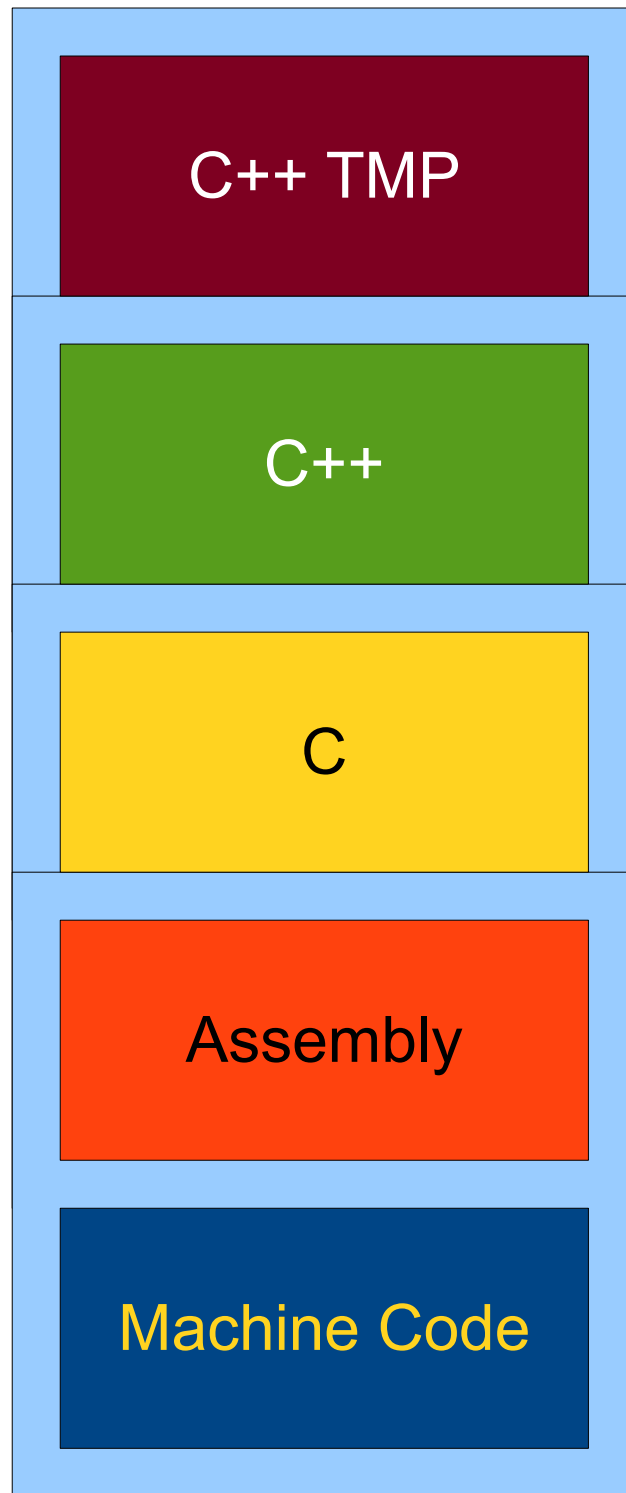


A **metaprogram** is a program that produces or manipulates constructs of a target language.

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**

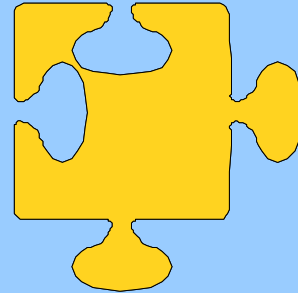
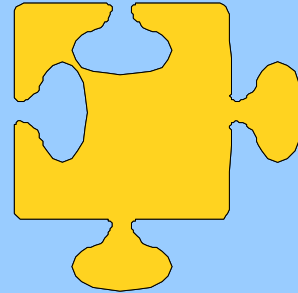
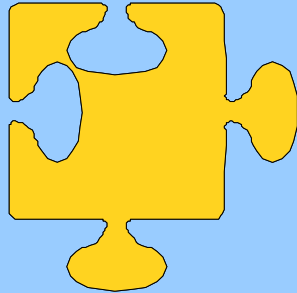
```
template <typename T> class Vector
{
public:
    /* ... ctors, dtor, etc. */
        T& operator[] (size_t);
    const T& operator[] (size_t) const;

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

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

# Vector

Type T



Range Checking

Logging

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 LoggingPolicy>  
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 LoggingPolicy>  
class Vector: public RangePolicy,  
             public LoggingPolicy  
{  
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 out_of_bounds("Bad!");
    }
};
```

# Another Sample Range Policy

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

# Another Sample Range Policy

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

# Yet Another Range Policy

```
class NoErrorPolicy
{
protected:
    ~NoErrorPolicy();
    void CheckRange(size_t pos,
                   size_t numElems)
    {
        // Do nothing
    }
};
```

# Implementer Code

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

# Implementer Code

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

# Implementer Code

```
template <typename T,  
         typename RangePolicy,  
         typename LoggingPolicy>  
T& Vector<T, RangePolicy, LoggingPolicy>::  
    operator[] (size_t position)  
{  
    LoggingPolicy::Log("Element lookup");  
    RangePolicy::CheckRange(position,  
                            this->size);  
    return this->elems[position];  
}
```



# Client Code

```
int main()
{
    Vector<int, ThrowingErrorPolicy,
        NoLoggingPolicy> 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 Specialization

- A version of a template to use when a specific pattern of arguments is 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  
};
```

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

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

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



# Random-Access Iterators

```
itr += distance;   itr + distance  
itr1 < itr2;     itr[myIndex]
```

## Bidirectional Iterators

```
--itr;
```

## Forward Iterators

### Input Iterators

```
val = *itr;  
++itr;
```

### Output Iterators

```
*itr = val;  
++itr;
```

```
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);
}
```

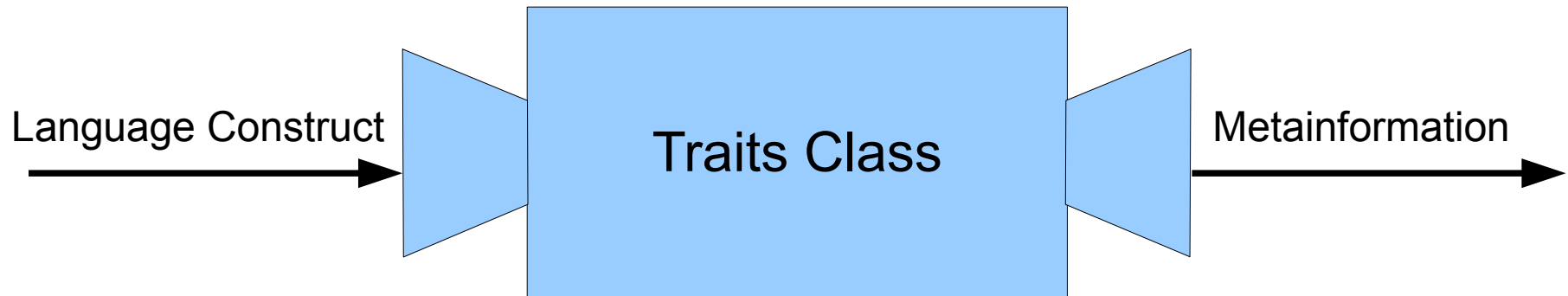
```
template <...>
    template <typename Iter>
void Vector<...>::insert(iterator where,
                        Iter start,
                        Iter stop)
{
    /* Insert elements one at a time. */
    for(; start != stop; ++start, ++where)
        where = insert(where, start);
}
```

```
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 :  
    public input_iterator_tag,  
    public output_iterator_tag {};  
  
struct bidirectional_iterator_tag :  
    public forward_iterator_tag {};  
  
struct random_access_iterator_tag :  
    public 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 iterator_traits<Iter>::iterator_category());
}
```

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

```

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

```

```

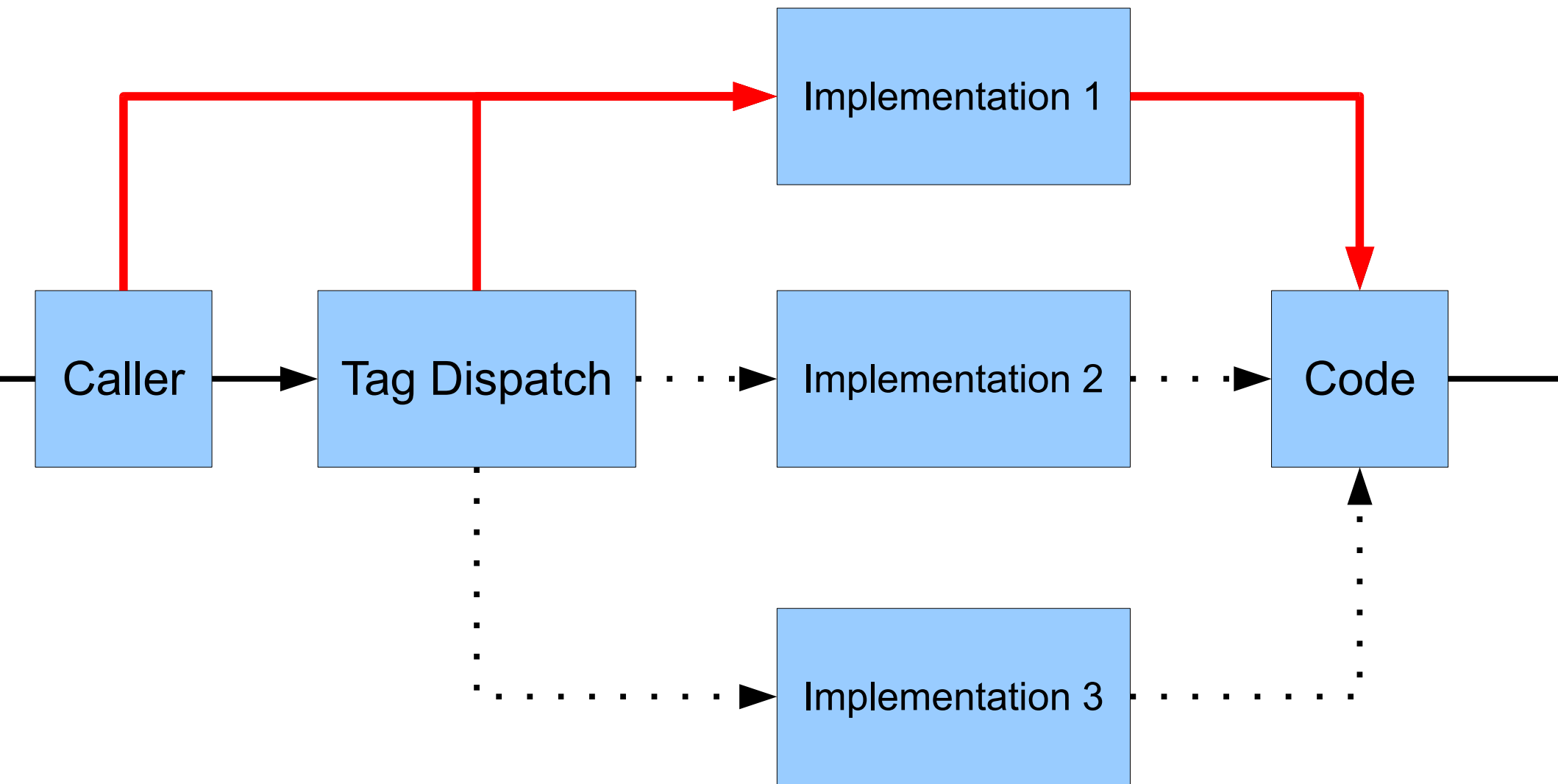
template <...>
    template <typename Iter>
void Vector<...>::doInsert(iterator where,
                          Iter start, Iter stop,
                          forward_iterator_tag)
{
    /* ... more complex logic to shift everything
     * down at the same time...
     */
}

```

```
template <...>
    template <typename Iter>
void Vector<...>::doInsert(iterator where,
                          Iter start, Iter stop,
                          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,
                          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)
```

# List Recursion with Templates

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

# List Recursion with Templates

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

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

# List 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 Engine { /* ... */ };  
  
class CarEngine: public Engine { /* ... */ };  
  
class DeLoreanEngine: public Engine { /* ... */ };  
  
class JetEngine: public Engine { /* ... */ };  
  
class GoldPlatedJetEngine: public JetEngine { /* ... */ };
```

```
class Brakes { /* ... */ };  
  
class CarBrakes: public Brakes { /* ... */ };  
  
class DeLoreanBrakes: public CarBrakes { /* ... */ };  
  
class JetBrakes: public Brakes { /* ... */ };  
  
class GoldPlatedJetBrakes: public JetBrakes { /* ... */ };
```

```
class Chassis { /* ... */ };  
class CarChassis: public Chassis { /* ... */ };  
class DeLoreanChassis: public Chassis { /* ... */ };  
class JetChassis: public Chassis { /* ... */ };  
class GoldPlatedJetChassis: public JetChassis { /* ... */ };
```

```
class VehicleFactory
{
public:
    virtual ~VehicleFactory() {}
    virtual Engine*   createEngine() = 0;
    virtual Brakes*   createBrakes() = 0;
    virtual Chassis*  createChassis() = 0;
};
```



```
class VehicleFactory
{
public:
    virtual ~VehicleFactory() {}
    virtual Engine* createEngine() = 0;
    virtual Brakes* createBrakes() = 0;
    virtual Chassis* createChassis() = 0;
};
```

```
class CarFactory: public VehicleFactory
{
public:
    virtual CarEngine* createEngine()
{ return new CarEngine; }
    virtual CarBrakes* createBrakes()
{ return new CarBrakes; }
    virtual CarChassis* createChassis();
{ return new CarChassis; }
};
```

```
class CPU { /* ... */ };  
class IntelCPU: public CPU { /* ... */ };  
class AMDCPU: public CPU { /* ... */ };  
class AtmelCPU: public CPU { /* ... */ };
```

```
class RAM { /* ... */ };  
  
class DDRRAM: public RAM { /* ... */ };  
  
class DDR2RAM: public RAM { /* ... */ };  
  
class DDR3RAM: public RAM { /* ... */ };
```

```
class HardDrive { /* ... */ };  
  
class SATADrive: public HardDrive { /* ... */ };  
  
class SCSIIDrive: public HardDrive { /* ... */ };  
  
class RAIDDrive: public HardDrive { /* ... */ };
```

```
class ComputerFactory
{
public:
    virtual ~ComputerFactory() {}
    virtual CPU*      createCPU() = 0;
    virtual RAM*     createRAM() = 0;
    virtual HardDrive* createHardDrive() = 0;
};
```

Can we automatically generate abstract factories?

Yes!

Idea One: Rewrite Using Tag Dispatching



# A Deceptively Simple Class

```
template <typename T> struct Box {};
```

```
class ComputerFactory
{
public:
    virtual ~ComputerFactory() {}
    virtual CPU*      createCPU() = 0;
    virtual RAM*     createRAM() = 0;
    virtual HardDrive* createHardDrive() = 0;
};
```

```
class ComputerFactory
{
public:
    virtual ~ComputerFactory() {}
    virtual CPU*      createImpl (Box<CPU>) = 0;
    virtual RAM*     createImpl (Box<RAM>) = 0;
    virtual HardDrive* createImpl (Box<HardDrive>) = 0;
};
```

```
class ComputerFactory
{
public:
    virtual ~ComputerFactory() {}
    virtual CPU*      createImpl(Box<CPU>) = 0;
    virtual RAM*     createImpl(Box<RAM>) = 0;
    virtual HardDrive* createImpl(Box<HardDrive>) = 0;
    template <typename T> T* create();
};
```

```
template <typename T> T* ComputerFactory::create()
{
    return createImpl(Box<T>());
}
```

```
class ComputerFactory
{
public:
    virtual ~ComputerFactory() {}
    virtual CPU*      createImpl(Box<CPU>) = 0;
    virtual RAM*     createImpl(Box<RAM>) = 0;
    virtual HardDrive* createImpl(Box<HardDrive>) = 0;
    template <typename T> T* create();
};
```

```
template <typename T> T* ComputerFactory::create()
{
    return createImpl(Box<T>());
}
```

```
// Client use:  
myFactory->create<CPU>();
```

Idea Two: Parameterize the class over a **typelist** of the types to create.

```
template <typename> class FactoryImpl;
```

```
template <typename> class FactoryImpl;
```

```
template <typename T> class FactoryImpl<LIST1(T)>  
{  
public:  
    virtual ~FactoryImpl() {};  
    virtual T* createImpl(Box<T>) = 0;  
}
```

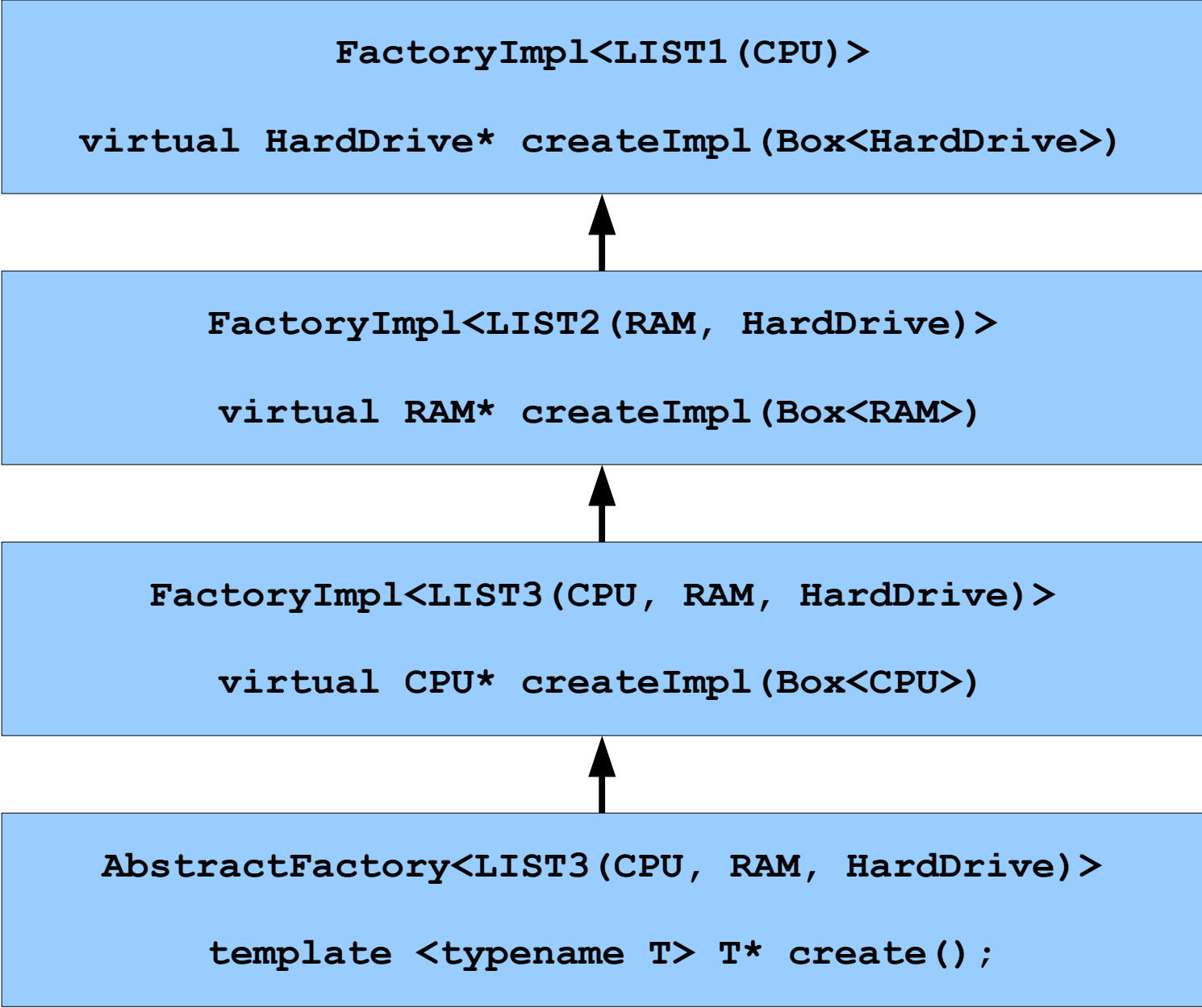


```
template <typename> class FactoryImpl;
```

```
template <typename T> class FactoryImpl<LIST1(T)>  
{  
public:  
    virtual ~FactoryImpl() {};  
    virtual T* createImpl(Box<T>) = 0;  
}
```

```
template <typename T, typename Tail>  
class FactoryImpl<Cons<T, Tail> >: public FactoryImpl<Tail>  
{  
public:  
    virtual T* createImpl(Box<T>) = 0;  
    using FactoryImpl<Tail>::createImpl;  
};
```

```
template <typename Types>
class AbstractFactory: public FactoryImpl<Types>
{
public:
    template <typename T> T* create()
    {
        return createImpl(Box<T>());
    }
};
```



```
AbstractFactory<LIST3 (CPU,  
RAM, HardDrive)>
```

```
virtual CPU*          createImpl (Box<CPU>)  
virtual RAM*         createImpl (Box<RAM>)  
virtual HardDrive*   createImpl (Box<HardDrive>)  
template <typename T> T* create()
```

A word of warning...

# **The Limits of Template Metaprogramming**

A Turing machine can simulate C++ templates.

C++ templates can simulate Turing machines.

**C++ templates are Turing-complete.**

In other words, C++ templates have 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?**