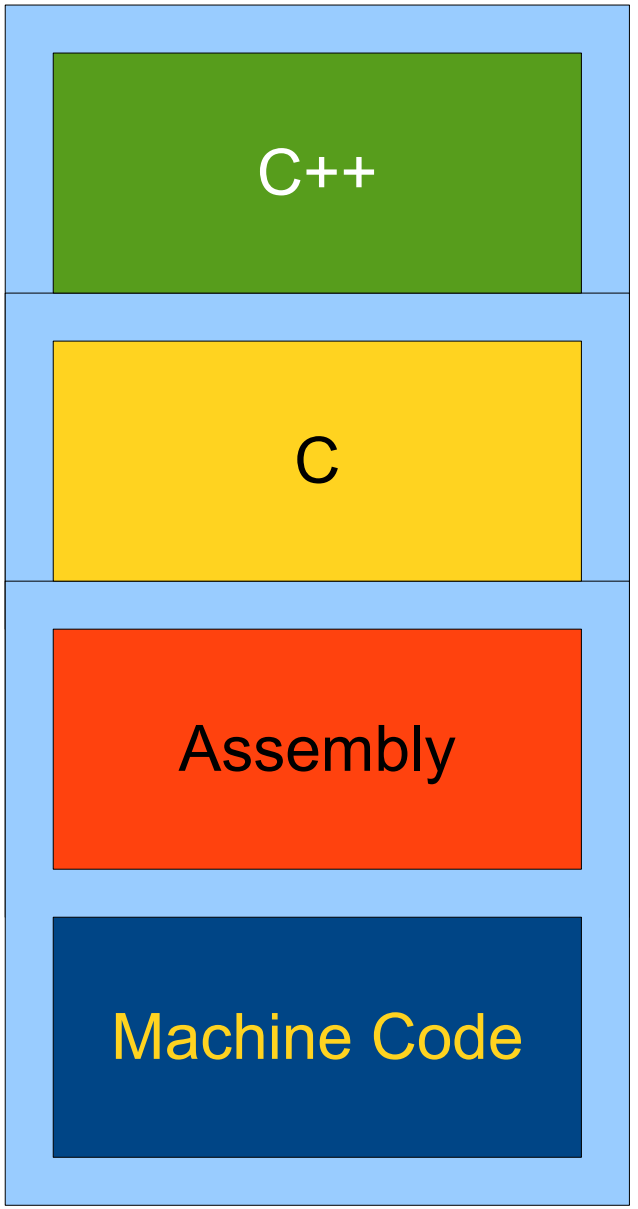
C++ Template Metaprogramming in 15 Minutes

C++ Template Metaprogramming in 15ish Minutes

A **program** is a sequence of instructions which creates and modifies **data**.

A **metaprogram** is a sequence of instructions which creates and modifies **programs**.

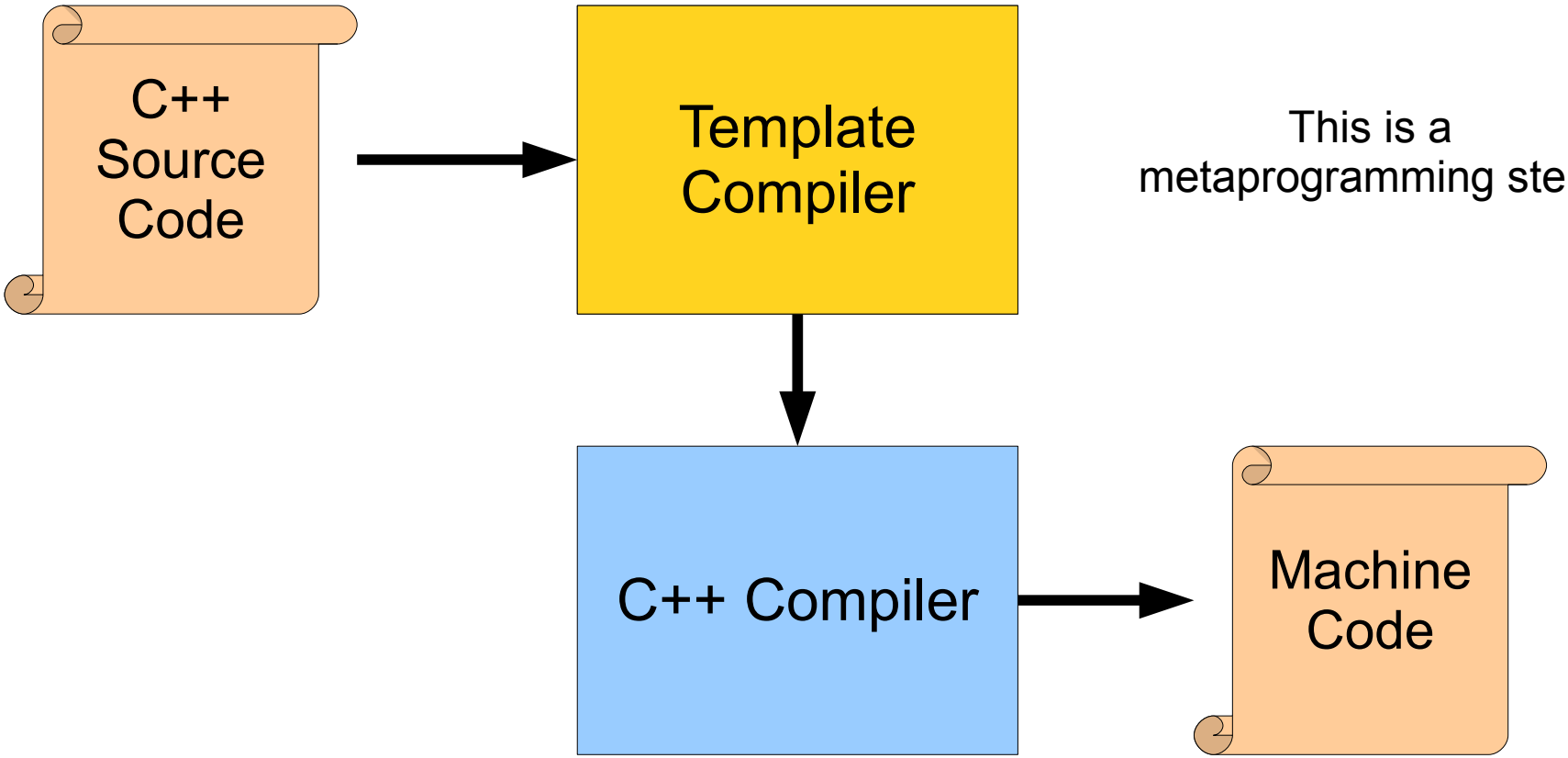
Why would you ever want to do this?


C++
C
Assembly
Machine Code

Metaprogramming introduces
new abstractions into the host language.

So what exactly is
template metaprogramming?

How C++ Templates Work

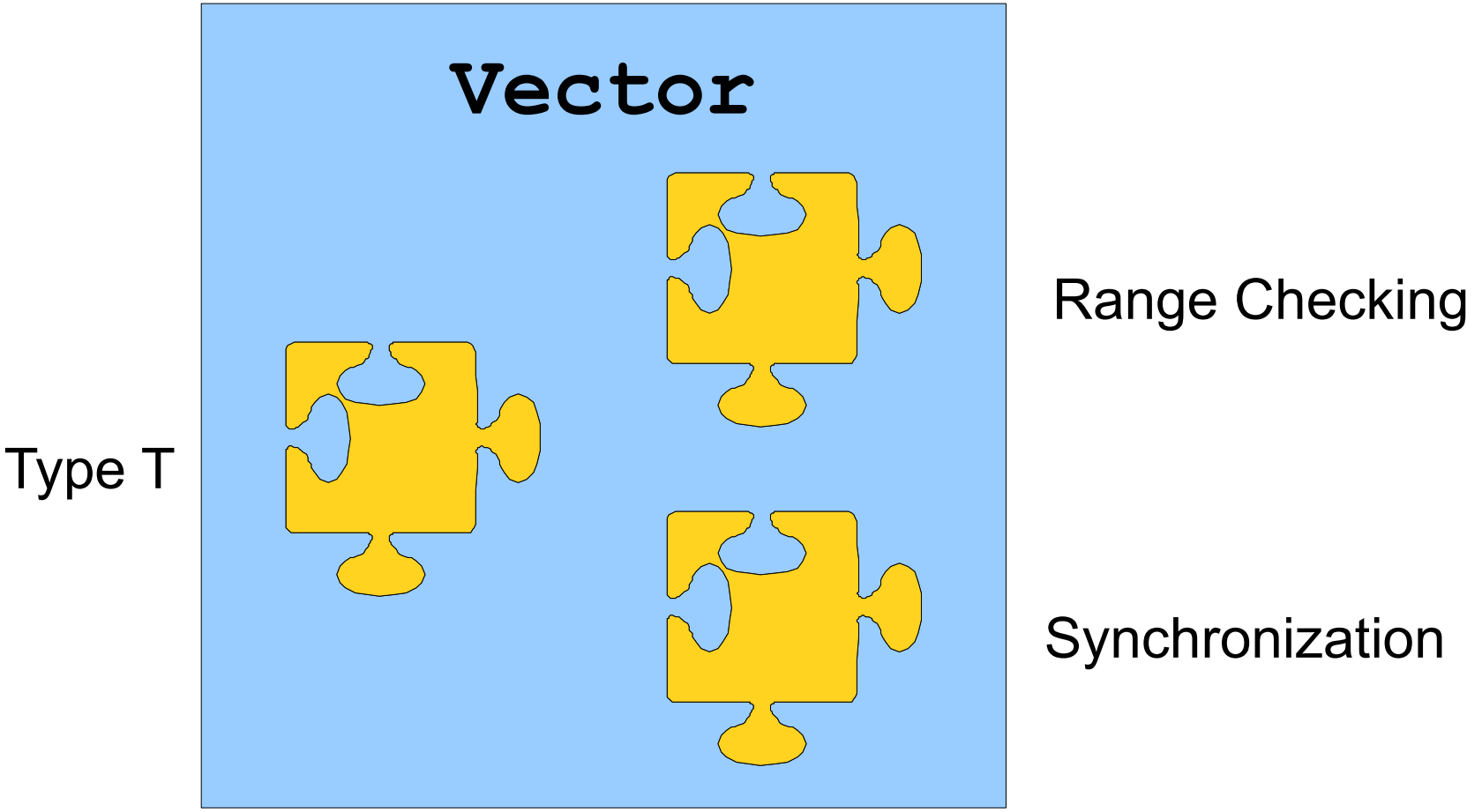


A **template metaprogram** is a C++ program that uses templates to generate customized C++ code at compile-time.

A new abstraction: **policies**.

```
template <typename T> class Vector
{
public:
   T getAt(int index);
   void setAt(int index, T value);

   /* ... 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:
   T getAt(int index);
   void setAt(int index, T value);

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

```
template <typename T,
          typename RangePolicy,
          typename LockingPolicy>
class Vector
{
public:
   T getAt(int index);
   void setAt(int index, T value);

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

```
template <typename T,
          typename RangePolicy,
          typename LockingPolicy>
class Vector: public RangePolicy,
              public LockingPolicy
{
public:
   T getAt(int index);
   void setAt(int index, T value);

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

Sample Range Policy

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

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

Another Sample Range Policy

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

Another Sample Range Policy

```
class LoggingErrorPolicy
{
public:
    void setLogFile(std::string filename);
protected:
    ~LoggingErrorPolicy();
    void CheckRange(int pos,
                    int 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>::
      getAt (int position)
{



   return this->elems[position];
}
```

Implementer Code

```
template <typename T,
          typename RangePolicy,
          typename LockingPolicy>
T Vector<T, RangePolicy, LockingPolicy>::
      getAt (int position)
{
   LockingPolicy::Lock lock;


   return this->elems[position];
}
```

Implementer Code

```
template <typename T,
          typename RangePolicy,
          typename LockingPolicy>
T Vector<T, RangePolicy, LockingPolicy>::
      getAt (int 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;
}
```

Or this...

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

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

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

   return 0;
}
```

Or even this...

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

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

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

   return 0;
}
```

Policy classes **beat the combinatorial explosion** of possible implementations.

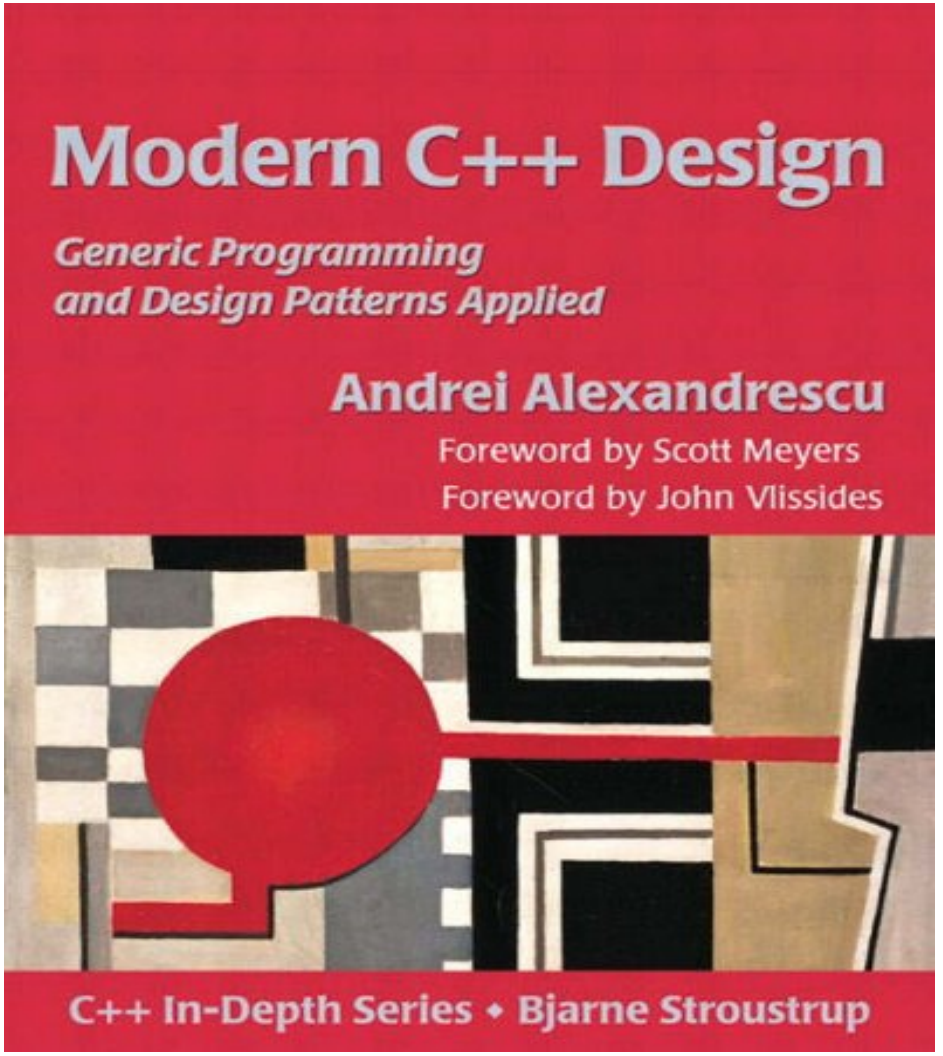
This is **not possible** without template metaprogramming.

But this is **just a taste** of what's possible with template metaprogramming.

Template Metaprogramming Does:

- Compile-time dimensional analysis.
- Multiple dispatch.
- Design patterns.
- Code optimization.
- Lexing and parsing.
- *And so much more...*

Recommended Reading



- More template metaprogramming than you'll know what to do with.
- Very advanced material; be prepared to be overwhelmed!
- Considered the seminal work in modern C++ programming.