A Problem With Coordination

This item looks at problem in interface design that plays off safety and flexibility. Let's look at a class template that is parameterized with some sort of container. For example, we could design a generic stack whose implementation can be any sequential container with the appropriate member functions and semantics.

```
template <typename T, class Cont>
class Stack {
  public:
    Stack();
    ~Stack();
    void push( const T & );
    //…
  private:
    Cont s;
};
```

A user of the stack has to provide two template arguments, an element type and a container type, and the container has to be able to hold objects of the element type.

```
Stack<int, List<int> > aStack1; // OK
Stack<double, List<int> > aStack2; // legal, not OK
Stack<std::string, Deque<char *> > aStack3; // error!
```

We have a potential problem in coordination. If the user selects the incorrect type of container for the element type, we'll get a compile time error or a subtle bug. Additionally, most users of `Stack` don't really want to be bothered with selection of its underlying implementation, and will be satisfied with a reasonable default. We can improve the situation by providing a common default for the second template parameter.

```
template <typename T, class Cont = List<T> >
class Stack {
  //...
};
```

This helps in cases where the user of a `Stack` is willing to accept a `List` implementation, or doesn't particularly care about the implementation.

```
Stack<int> aStack1; // container is List<int>
Stack<double> aStack2; // container is List<double>
```

This is more or less the approach employed by the standard container adapters `stack`, `queue`, and `priority_queue`.

```
std::stack<int> stds;
        // container is std::deque< int,std::allocator<int> >
```

This approach is a good compromise of convenience for the casual user of the `Stack` facility, and of maximal flexibility for the experienced user to employ any (legal and effective) kind of container to hold the `Stack`'s elements.

However, this flexibility comes at a cost in safety. It's still necessary to coordinate the types of element and container in other instantiations, and this requirement of coordination opens up the possibility of mis-coordination.

```
Stack<int, Deque<int> > aStack3;
Stack<int, List<unsigned> > aStack4; // oops!
```

Let's see if we can improve safety and still have reasonable flexibility. A class template can also take a parameter that is itself a template. These parameters have the pleasingly repetitious name of template template parameters.

```
template <typename T, template <typename> class Cont>
class Stack {
  public:
    Stack();
    ~Stack();
    void push( const T & );
    //…
  private:
    Cont<T> s;
};
```

This approach allows coordination between element and container to be handled by the implementation of the `Stack` itself, rather than in all the various code that instantiates `Stack`. This single point of instantiation reduces the possibility of mis-coordination between the element type and the container used to hold the elements.

```
Stack<int,List> aStack1;
Stack<std::string,Deque> aStack2;
```

For additional convenience, we can employ a default for the template template argument.

```
template <typename T, template <typename> class Cont = Deque>
class Stack {
  public:
    Stack();
    ~Stack();
    void push( const T & );
    //…
  private:
    Cont<T> s;
};
//…
Stack<int> aStack1; // container is Deque<int>
Stack<std::string,List> aStack2; // container is List<string>
```

This is often a good approach for dealing with coordination of a set of arguments to a template and a template that is to be instantiated with the arguments. However, this gain in safety and simplicity comes at the cost of some loss of flexibility. For instance, it would not be unreasonable to want to instantiate our `Stack` with a container that could hold the elements without loss of information.

```
Stack< short,Deque<int> > aStack3; // error! need template, not type
```

Unfortunately, this approach forces us to choose between use of a template or a typename to specify the container. An incrementally more flexible approach might employ a third template parameter and default.

```
template <typename T,
              template <typename> class Cont = Deque,
              typename CT = T>
class Stack {
  public:
    Stack();
    ~Stack();
    void push( const T & );
    //…
  private:
    Cont<CT> s;
};
```

We now have the ability to be fairly specific, but still have convenience and coordination by default.

```
Stack<int> s1; // container is Deque<int>
Stack<int,List> s2; // container is List<int>
Stack<short,Deque,int> s3; // container is Deque<int>
Stack<int,List,unsigned> s3; // oops! container is List<unsigned>
```

However, we still have not achieved the flexibility of the standard approach. We still have a problem if we want to instantiate a `Stack` with a container that is not generated from a template, or is generated from a template that accepts something other than a single typename argument. For example, we are not able to instantiate `Stack` with any of the standard containers.

```
Stack<int,MyIntContainer> s4; // error! need template, not type
Stack<int,std::deque> s5; // error! need 1-parm template
```

The standard sequential containers take two type parameters (for the element type and the allocator type). That the allocator type has a default does not help us in this case.[1]

[1] One interesting note: It's quite easy to get information about a template from a type instantiated from that template (see "Once, Weakly" for 6 January 2003). However, it is quite difficult (or impossible?) to get information about a template name if it is uninstantiated.