Date:.....

Item 25: Consider support for a non-throwing swap.

Implementation of std::swap.

```
namespace std {

  template < typename T >
  void swap ( T& a, T& b)
    {
      T temp(a);
      a = b;
      b = temp;
    }
}
```

→ As long as your type support copying (via copy ctor & copy assignment operator) default swap will work.

→ But default swap puts you on fast track to slow lane by copying a to temp, b to a, temp to b.

→ We can use pimpl idiom (pointer to implementation)

Date:.....

```
class WidgetImpl {
    public:
    ...
    private:
      int a, b, c;
      std::vector<double> v;
      ...
  };

  class Widget {  //class using pimpl idiom
public:
    Widget (const Widget & rhs);

  Widget & operator=(const Widget & rhs)
      { ...
          *pImpl = *(rhs.pImpl);
          ...
      }
    private:
          WidgetImpl * pImpl;
  };
```

=> Default swap will not only copy 3 widget objects but also 3 WidgetImpl object. which is very inefficient.

Date:

we would like to tell std::swap, that we want to swap their internal pimpl ptrs.

i.e.

```
namespace std {
    template <>
    void swap<Widget>(Widget& a,
                      Widget& b)
    {
        swap(a.pImpl, b.pImpl);
    }
} //won't compile
```

coz its trying to access the pimpl pointers inside a and b & they're private.

template <> at the beginning of this functn says that this is a "total template specializatn" for std::swap & <wiget> after the name of functn says specializatn is when T is Widget.

Date:.....

→ we are allowed to specialize standard templates (like swap) for types of own creatn (such as widget).

→ We want to have Widget declare a public member fncn swap that does actual swapping & specialize std::swap to call the member fncn.

```
class Widget {
 public:
     void swap( Widget & other)
     {
       using std::swap;
       swap( pImpl, other.pImpl);
     }
};

namespace std {

  template <>
  void swap<Widget> (Widget& a,
                     Widget & b)
     {
       a.swap(b);
     }
}
```

Date:.....

Suppose Widget & WidgetImpl are class templates.

```
template <typename T>
class WidgetImpl {...};

template <typename T>
class Widget {...};
```

⇒ we run into trouble with specializatn for std::swap.

```
namespace std {

  template <typename T>
  void swap <Widget<T>>(Widget<T>& a,
                        Widget<T> & b) // illegal code.
    { a.swap(b); }
}
```

⇒ we are trying to partially specialize a funcn template (std::swap), though C++ allows partial specializatn of class templates, it does not allow it for funcn templates.

Date:.....

Or else we can add an overload

```
namespace std {

    template <typename T>
    void swap (Widget<T> & a, Widget<T> & b)

        { a.swap(b); } //overloading of
                        std::swap
```

=> In general overloading funcn templates is fine but std is a special namespace and rules governing it are also special.

-> It is fine to totally specialize templates in std, but adding new template is not okay.

Programs that cross this line may compile but they will have undefined behaviour.

=> What to do now?

We still declare a non-member swap that calls the member swap but we don't declare non-member swap to be specializatn of std::swap

Date:.....

Instead we use different namespace

```
namespace WidgetStuff
{
    template <typename T>
    class Widget {...};

    template <typename T>
    void swap (Widget<T>& a,
               Widget<T>& b)
    { a.swap(b); }   // non member swap
                        funcn, not part
                        of std::swap
}
```

⇒ Which swap should below code call?

```
template <typename T>
void doSomething (T& obj1, T& obj2)
{
    swap(obj1, obj2);
}
```

One in std or specializatn of general
One in std or T specific one.

Date:.....

```
template<typename T>
void dosomething(T& obj1, T& obj2)
{ using std::swap;
    swap(obj1, obj2);
}
```

→ When compilers see the call to swap, they search for the right swap to invoke.

→ Compilers will prefer T-specific specialization of std::swap over general template

```
&
std::swap(obj1, obj2);
```

//wrong way to call swap since it will force the compiler to use swap in std.