STL Function Adapters
STL Function Adapters

- STL has predefined functor adapters that will change their functors so that they can:
  - Perform function composition & binding
  - Allow fewer created functors
STL Function Adapters

- These functors allow programs to combine, transform, or manipulate functors with each other, certain values, or with special functions

```cpp
vector<const char *> aVect { "One", "Two", "Three", "Four", "Five" }

auto itr = find_if(aVect.begin(), aVect.end(),
    not1(bind2nd(ptr_fun(strcmp), "Three")));
```
STL Function Adapters

• There are three types of STL function adapters
There are three types of STL function adapters:

- Binders (`bind1st()`, `bind2nd()`, & `bind()`) bind one of their arguments.

```cpp
vector<double> myVector{10.1, 20.2, 30.3, 40.4, 50.5};
auto n = count_if(myVector.begin(), myVector.end(),
                 bind(greater<>(), _1, value));
```
STL Function Adapters

- There are three types of STL function adapters
  - Binders (bind1st(), bind2nd(), & bind()) bind one of their arguments
  - Negators (not1, not2, & not_fn) adapt functors by negating arguments

```cpp
vector<int> v{4, 1, 2, 8, 5, 7};
auto itr = find_if(v.begin(), v.end(), not_fn(bind(greater<> (), _1, 3)));
```
STL Function Adapters

- There are three types of STL function adapters
  - Binders (bind1st(), bind2nd(), & bind()) bind one of their arguments
  - Negators (not1, not2, & not_fn) adapt functors by negating arguments
  - Member functions (ptr_fun, mem_fun, & mem_fn) allow functors to be class members

```cpp
template<typename T> class vector { ... }

vector<const char *> aVect { "One", "Two", "Three", "Four", "Five" };

auto itr = find_if(aVect.begin(), aVect.end(),
    not1(bind2nd
        (ptr_fun
            (strcmp),
            "Three")));
```
STL Binder Function Adapter

- A binder can be used to transform a binary functor into a unary one by acting as a converter between the functor & an algorithm
STL Binder Function Adapter

- Binders always store both the binary functor & the argument internally (the argument is passed as one of the arguments of the functor every time it is called)
  - `bind1st(op, arg)` calls ‘op’ with ‘arg’ as its first parameter
  - `bind2nd(op, arg)` calls ‘op’ with ‘arg’ as its second parameter
  - `bind(op, placeholders, arg)` calls ‘op’ with ‘arg’ as its first or second parameter

```cpp
template<typename _Operation, typename _Tp>
bind2nd<_Operation, bind2nd>(const _Operation& __fn, const _Tp& __x) {
    typedef typename _Operation::second_argument_type _Arg2_type;
    return binder2nd<_Operation>(__fn, _Arg2_type(__x));
}
```
void contrast_bind1st_bind2nd_and_bind() {
    auto p1 = bind1st(plus<int>(), 10);
    auto p2 = bind2nd(plus<int>(), 10);
    auto p3 = bind(plus<int>(), 10, placeholders::_1);
    auto p4 = bind(plus<int>(), placeholders::_1, 10);
    cout << p1(20) << endl; cout << p2(20) << endl;
    cout << p3(20) << endl; cout << p4(20) << endl;

    auto m1 = bind1st(minus<int>(), 10);
    auto m2 = bind2nd(minus<int>(), 10);
    auto m3 = bind(minus<int>(), 10, placeholders::_1);
    auto m4 = bind(minus<int>(), placeholders::_1, 10);
    cout << m1(20) << endl; cout << m2(20) << endl;
    cout << m3(20) << endl; cout << m4(20) << endl; ...
}

See github.com/douglascraigschmidt/CPlusPlus/tree/master/STL/S-09/9.2