```
/*
 * Function Objects (functors)
 *
 * Example:
 */
class X {
   public:
   void operator()(string str) {
      cout << "Calling functor X with parameter " << str<< endl;
   }
};

int main()
{
   X foo;
   foo("Hi");    // Calling functor X with parameter Hi
}
/*
 * Benefits of functor:
 * 1. Smart function: capabilities beyond operator()
 *    It can remember state.
 * 2. It can have its own type.
 */

//
//   operator string () const { return "X"; }

/*
 * Benefits of functor:
 * 1. Smart function: capabilities beyond operator()
 *    It can remember state.
 * 2. It can have its own type.
 */

/*
 * Parameterized Function
 */
class X {
   public:
   X(int i) {}
   void operator()(string str) {
      cout << "Calling functor X with parameter " << str<< endl;
   }
};

int main()
{
   X(8)("Hi");
}
```

```
void add2(int i) {
   cout << i+2 << endl;
}

template<int val>
void addVal(int i) {
   cout << val+i << endl;
}

class AddValue {
   int val;
   public:
   AddValue(int j) : val(j) { }
   void operator()(int i) {
      cout << i+val << endl;
   }
};

int main()
{
   vector<int> vec = { 2, 3, 4, 5};
   //for_each(vec.begin(), vec.end(), add2); // {4, 5, 6, 7}
   int x = 2;
   //for_each(vec.begin(), vec.end(), addVal<x>); // {4, 5, 6, 7}
   for_each(vec.begin(), vec.end(), AddValue(x)); // {4, 5, 6, 7}
}

/* Notes:
//global variable: int val;

template<int val>
void addVal(int i) {
   cout << val+i << endl;
}
//std::for_each(vec.begin(), vec.end(), addVal<3>);

class AddValue {
   int val;
   public:
   AddValue(int j) : val(j) { }
   void operator()(int i) {
      cout << i+val << endl;
   }
};
//int x = 9;
//std::for_each(vec.begin(), vec.end(), AddValue(x));
*/
```

```
/*
 * Build-in Functors
 */
less greater  greater_equal  less_equal  not_equal_to
logical_and  logical_not  logical_or
multiplies minus  plus  divide  modulus  negate

int x = multiplies<int>()(3,4);  //  x = 3 * 4

if (not_equal_to<int>()(x, 10))   // if (x != 10)
   cout << x << endl;

/*
 * Parameter Binding
 */
set<int> myset = { 2, 3, 4, 5};
vector<int> vec;


int x = multiplies<int>()(3,4);  //  x = 3 * 4

// Multiply myset's elements by 10 and save in vec:
transform(myset.begin(), myset.end(),    // source
            back_inserter(vec),              // destination
                 bind(multiplies<int>(), placeholders::_1, 10));  //
functor
    // First parameter of multiplies<int>() is substituted with myset's
element
    // vec: {20, 30, 40, 50}


void addVal(int i, int val) {
   cout << i+val << endl;
}
for_each(vec.begin(), vec.end(), bind(addVal, placeholders::_1, 2));

// C++ 03: bind1st, bind2nd

void addVal(int i, int val) {
   cout << i+val << endl;
}
for_each(vec.begin(), vec.end(), bind(addVal, placeholders::_1, 2));

// C++ 03: bind1st, bind2nd

// Convert a regular function to a functor
double Pow(double x, double y) {
      return pow(x, y);
}
```

```
int main()
{
  set<int> myset = {3, 1, 25, 7, 12};
  deque<int> d;
  auto f = function<double (double,double)>(Pow);   //C++ 11
  transform(myset.begin(), myset.end(),     // source
                back_inserter(d),              // destination
                     bind(f, placeholders::_1, 2));  // functor
           //  d: {1, 9, 49, 144, 625}
}
// C++ 03 uses ptr_fun

set<int> myset = {3, 1, 25, 7, 12};
// when (x > 20) || (x < 5),  copy from myset to d
deque<int> d;

bool needCopy(int x){
   return (x>20)||(x<5);
}

transform(myset.begin(), myset.end(),     // source
          back_inserter(d),               // destination
          needCopy
          );

// C++ 11 lambda function:
transform(myset.begin(), myset.end(),     // source
          back_inserter(d),              // destination
          [](int x){return (x>20)||(x<5);}
          );

/*
          bind(logical_or<bool>,
              bind(greater<int>(), placeholders::_1, 20),
              bind(less<int>(), placeholders::_1, 5))

// C++ 11 lambda function:
transform(myset.begin(), myset.end(),     // source
          back_inserter(d),              // destination
          [](int x){return (x>20)||(x<5);}
          );

bool needCopy(int x){
   return (x>20)||(x<5);
}
*/

/*
 * Why do we need functor in STL?
 *
 */

set<int> myset = {3, 1, 25, 7, 12}; // myset: {1, 3, 7, 12, 25}
```

```
// same as:
set<int, less<int> > myset = {3, 1, 25, 7, 12};

bool lsb_less(int x, int y) {
      return (x%10)<(y%10);
}

class Lsb_less {
   public:
   bool operator()(int x, int y) {
      return (x%10)<(y%10);
   }
};
int main()
{
  set<int, Lsb_less> myset = {3, 1, 25, 7, 12};  // myset: {1,12,3,25,7}
  ...
}

/* Notes

bool lsb_less(int x, int y) {
      return (x%10)<(y%10);
}

class Lsb_less {
   public:
   bool operator()(int x, int y) {
      return (x%10)<(y%10);
   }
};
*/

/*
 * Predicate
 *
 * A functor or function that:
 * 1. Returns a boolean
 * 2. Does not modify data
 */

class NeedCopy {
   bool operator()(int x){
      return (x>20)||(x<5);
   }
};

transform(myset.begin(), myset.end(),     // source
          back_inserter(d),               // destination
          NeedCopy()
          );

// Predicate is used for comparison or condition check
// More About Functors
```