• Programming treats computation as the evaluation of pure mathematical functions
  – First-class functions: functions can be stored, copied, etc
  – Closures: functions remember environment at which it was created
  – Higher-order functions: functions take functions as parameters
  – Lambda functions: anonymous functions
  – Pure functions: functions cannot have mutable state
  – Function composition: output of one function is input to another
  – Currying: chaining functions each with one parameter
  – Recursion: without mutable state, need recursion to repeat

**Develop a generic count algorithm**

Develop a generic count function that as input a sequence ADT (seq) and a value to search for in the array. The return type of the function is int. It should return a count of the number of elements in the sequence that match the given value. The function should be generic across any kind of sequence which might store any type of values. In other words, the function should work for a List<int>, a List<float>, a Vector<int>, etc. *Hint: Develop a version of the algorithm specialized for a List<int> and then make it generic.*
• Generic over the sequence, specialized for a given \textit{predicate}
• Can only check for equality
• Can we make this function parameterized by the predicate?
• Pass in a function pointer to use for testing the predicate

\textbf{Using C function pointers}

```
bool threshold_25( int v ) { return ( v > 25 ); }

typedef bool ( *pred_func_t )( int );

template < typename S >
int count_if( const S& seq, pred_func_t pred )
{
    int count = 0;
    for ( auto v : seq )
        if ( pred(v) )
            count++;
    return count;
}

int main( void )
{
    List list;
    list.push_front( 12 );
    list.push_front( 15 );
    list.push_front( 50 );
    list.push_front( 06 );
    list.push_front( 76 );
    list.push_front( 37 );

    int a = count_if( list, &threshold_25 );
    return 0;
}
```
1. C++ Functors

- Use object-oriented and generic programming to implement:
  - First-class functions: objects will act like functions
  - Closures: environment will be explicitly stored in object
  - Higher-order functions: functions can be generic over functor parameters

```cpp
class Threshold
{
    public:

    Threshold( int threshold )
        : m_threshold( threshold )
    {
    }

    bool call( int v ) const
    {
        return ( v > m_threshold );
    }

    private:
        int m_threshold;
};

int main( void )
{
    int x = 25; // environment for functor
    auto pred0 = Threshold(x); // explicit closure
    auto pred1 = pred1; // copy a functor
    bool a = pred1.call( 15 ); // call a stored functor
    bool b = pred1.call( 30 ); // call a stored functor
}
```
• Overload the call operator to enable true “function-call” syntax

```cpp
class Threshold {
public:
    Threshold( int threshold ) : m_threshold( threshold ) {
    }

    bool operator()( int v ) const {
        return ( v > m_threshold );
    }

private:
    int m_threshold;
};

int main( void ) {
    int x = 25;          // environment for functor
    auto pred0 = Threshold(x); // explicit closure
    auto pred1 = pred1;    // copy a functor
    bool a = pred1( 15 ); // call a stored functor
    bool b = pred1( 30 ); // call a stored functor
}
```
• Use templates to make algorithms generic over functors

```cpp
bool threshold_25( int v ) { return ( v > 25 ); }

template < typename S, typename Pred >
int count_if( const S& seq, Pred pred )
{
    int count = 0;
    for ( auto v : seq )
        if ( pred(v) )
            count++;
    return count;
}

int main( void )
{
    List list;
    list.push_front( 12 );
    list.push_front( 15 );
    list.push_front( 50 );
    list.push_front( 06 );
    list.push_front( 76 );
    list.push_front( 37 );

    int a = count_if( list, &threshold_25 );
    int b = count_if( list, Threshold(25) );
    return 0;
}
```