1 C++ Functors
2 C++ Lambdas
3 Drawing Framework Case Study
• Functional 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
  – Function composition: output of one function is input to another
  – Currying: chaining functions each with one parameter
  – Pure functions: functions cannot have mutable state
  – Recursion: without mutable state, need recursion to repeat
  – Strong underlying mathematical theory (lambda calculus)

**Develop a generic count algorithm**

Develop a generic count function that takes as input a sequence (seq), a value to search for, and returns the number of elements in the sequence that match the given value as an int. 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 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

Using C function pointers for generic count algorithm

```c
bool threshold_25( int x ) { return ( x > 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) ) // notice dereference is optional!
            count++;
    return count;
}

int main( void )
{
    List<int> lst;
    lst.push_front( 12 );
    lst.push_front( 15 );
    lst.push_front( 50 );
    lst.push_front( 06 );
    lst.push_front( 76 );
    int a = count_if( lst, &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

• Overload the call operator to enable true “function-call” syntax

```cpp
class Threshold25 {
public:
  bool call( int x ) const
  {
    return ( x > 25 );
  }
};

int main( void )
{
  // create a functor
  Threshold25 pred0();
  Threshold25 pred1 = pred0();
  // call a stored functor
  bool b = pred1.call( 15 );
  bool c = pred1.call( 30 );
}
```
# 1. C++ Functors

```cpp
class Threshold25 {
public:
    bool operator()( int x ) const {
        return ( x > 25 );
    }
};

int main( void ) {
    // create a functor
    Threshold25 pred0();

    // copy a functor
    Threshold25 pred1 = pred0;

    // call a stored functor
    bool b = pred1( 15 );

    // call a stored functor
    bool c = pred1( 30 );
}
```

- Use arrow pointing to the code to represent a functor with no state
1. C++ Functors

```cpp
class Threshold {
public:
    Threshold(int t) : m_t(t) {}

    bool operator()(int x) const {
        return (x > m_t);
    }

private:
    int m_t;
};

int main(void) {
    // create a functor
    Threshold pred0(25);

    // copy a functor
    Threshold pred1 = pred0;

    // call a stored functor
    bool b = pred1(15);

    // call a stored functor
    bool c = pred1(30);
}
```

- Functors can also be used to explicitly capture their environment when constructed to create a closure
- Use pointer to the object to represent a functor with state (just like any object)
• Use templates to make algorithms generic over function pointers and functors

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

class Threshold
{
    public:
    Threshold( int t ) : m_t( t ) { }
    bool operator()( int x ) const { return ( x > m_t ); }
    private:
    int m_t;
};

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<int> lst;
    lst.push_front( 12 );
    lst.push_front( 15 );
    lst.push_front( 50 );
    lst.push_front( 06 );
    lst.push_front( 76 );

    int a = count_if( lst, &threshold_25 );
    int b = count_if( lst, Threshold(25) );
    return 0;
}
```
class Threshold
{
public:
    Threshold( int t ) : m_t( t )
    {
    }

    bool operator()( int x ) const
    {
        return ( x > m_t );
    }

private:
    int m_t;
};

template <>
template <>
int count_if<int[2],Threshold>(
    const int[2]& seq,
    Threshold pred )
{
    int count = 0;
    for ( auto v : seq )
        if ( pred(v) )
            count++;
    return count;
}

int main( void )
{
    int arr[] = { 15, 35 };
    int a = 25;
    Threshold p(a);
    int b = count_if( arr, p );
    return 0;
}
2. C++ Lambdas

• Use new C++ syntax along with object-oriented and generic programming to implement:
  – **Lambdas**: create anonymous functors on the fly

```cpp
int main( void )
{
    int a = 25            // environment for functor

    // creates an anonymous functor that explicitly captures a
    auto pred0 = [a]( int x )
    {
        return x > a;
    };

    auto pred1 = pred0;    // copy a lambda
    bool b = pred1( 15 );  // call a stored lambda
    bool c = pred1( 30 );  // call a stored lambda
}
```

• Use `[]` to specify how to capture the environment
  – explicit list of variable names to capture
  – `=` captures all referenced variables by value
  – `&` captures all referenced variables by reference

```cpp
// lambda that implicitly captures a (by value)
auto pred0 = [=]( int x )
{
    return x > a;
};

// lambda that implicitly captures a (by reference)
auto pred0 = [&]( int x )
{
    return x > a;
};
```
• Use templates to make algorithms generic over function pointers, functors, and lambdas

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

class Threshold
{
    public:
        Threshold( int t ) : m_t( t ) { }
    bool operator()( int x ) const { return ( x > m_t ); }

    private:
    int m_t;
};

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<int> lst;
    lst.push_front( 12 );
    lst.push_front( 15 );
    lst.push_front( 50 );
    lst.push_front( 06 );
    lst.push_front( 76 );

    int a = count_if( lst, &threshold_25 );
    int b = count_if( lst, Threshold(25) );
    int c = count_if( lst, []( int x ) { return x > 25; } );
    return 0;
}
```
2. C++ Lambdas

```cpp
template <>
int count_if<int[2],__lambda0>(
    const int[2]& seq,
    __lambda_0 pred )
{
    int count = 0;
    for ( auto v : seq )
        if ( pred(v) )
            count++;
    return count;
}

int main( void )
{
    int arr[] = { 15, 35 };
    int a = 25;
    auto p = [=]( int v ) {
        return v > a;
    };
    int b = count_if( arr, p );
    return 0;
}
```
3. Drawing Framework Case Study

- Canvas
  - Display
  - Translation
  - Scale
  - Rotate
  - Draw

- Shape
  - ID
  - Set ID
  - Translate
  - Scale
  - Rotate
  - Draw

- Drawing
  - Group
  - Canvas
  - Add
  - Display

- Line
  - P1, P2
  - Translate
  - Scale
  - Rotate
  - Draw

- Point
  - X, Y
  - Translate
  - Scale
  - Rotate
  - Draw

- Triangle
  - P1, P2, P3
  - Translate
  - Scale
  - Rotate
  - Draw

- Group
  - List of Shapes
  - Translate
  - Scale
  - Rotate
  - Draw
• Create animations by repeatedly drawing and clearing screen

```c
int main( void )
{
    // Create a group of lines forming a star
    Group star;
    for ( int i = 0; i < 8; i++ )
        star.add( Line( Point(0,0), Point(0,3) ) % (i*45) );

    // Randomly place stars in a group
    Group stars;
    for ( int i = 0; i < 6; i++ ) {
        int x_offset = ( rand() % 30 ) - 15;
        int y_offset = ( rand() % 30 ) - 15;
        stars.add( star + Point( x_offset, y_offset ) );
    }

    // Make it snow
    for ( int i = 0; i < 80; i++ ) {
        // Clear the screen
        if ( i != 0 ) {
            for ( size_t j = 0; j < 33; j++ )
                printf("\x1b[A");
        }

        // Draw the snowflakes
        Drawing drawing;
        drawing.add( stars + Point(0,30-i) );
        drawing.display();

        // Wait between frames
        usleep(100000);
    }

    return 0;
}
```
3. Drawing Framework Case Study

- We can use functional programming to refactor the animation code creating a true animation framework

```cpp
template<typename DrawFrame>
void animate(int num_frames, DrawFrame draw_frame)
{
    for (int i = 0; i < num_frames; i++) {

        // Clear the screen
        if (i != 0) {
            for (size_t j = 0; j < 33; j++)
                printf("\x1b[A");
        }

        // Draw the frame
        Drawing drawing;
        draw_frame(i, &drawing);
        drawing.display();

        // Wait between frames
        usleep(100000);
    }
}

int main(void)
{
    ... 

    animate(80, [&](int i, Drawing* drawing_p) {
        drawing_p->add(stars + Point(0,30-i));
    });

    return 0;
}
```

https://repl.it/@cbatten/ece2400-T15-ex2

Topic 15: Functional Programming