

ECE 2400 Computer Systems Programming Fall 2021

Topic 15: Functional Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-28-21-28

1	C++ Functors	4
2	C++ Lambdas	9
3	Drawing Framework Case Study	12

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

-
- 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

```
1  bool threshold_25( int x ) { return ( x > 25 ); }
2
3  typedef bool (*pred_func_t) (int);
4
5  template < typename S >
6  int count_if( const S& seq, pred_func_t pred )
7  {
8      int count = 0;
9      for ( auto v : seq )
10         if ( pred(v) ) // notice dereference is optional!
11             count++;
12     return count;
13 }
14
15 int main( void )
16 {
17     List<int> lst;
18     lst.push_front( 12 );
19     lst.push_front( 15 );
20     lst.push_front( 50 );
21     lst.push_front( 06 );
22     lst.push_front( 76 );
23
24     int a = count_if( lst, &threshold_25 );
25     return 0;
26 }
```

1. C++ Functors

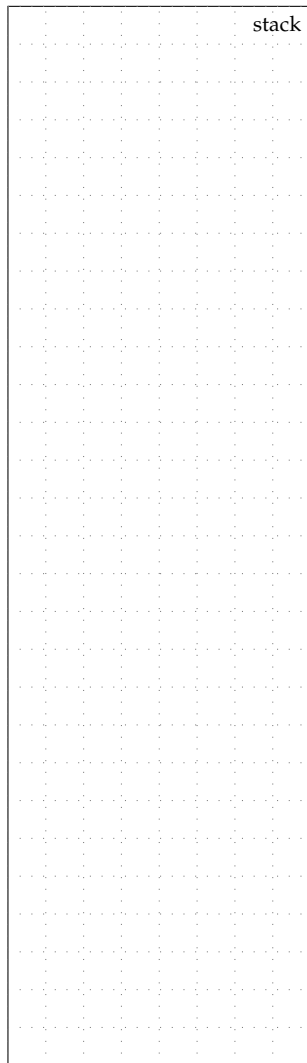
- Use object-oriented and generic programming to implement:
 - **First-class functions**: objects will act like functions
 - **Closures**: environment will be explicitly stored in object
 - **Higher-order functions**: functions can be generic over functor parameters
- Overload the call operator to enable true “function-call” syntax

```
1 class Threshold25
2 {
3     public:
4         bool call( int x ) const
5         {
6             return ( x > 25 );
7         }
8 };
9
10 int main( void )
11 {
12     // create a functor
13     Threshold25 pred0();
14
15     // copy a functor
16     Threshold25 pred1 = pred0;
17
18     // call a stored functor
19     bool b = pred1.call( 15 );
20
21     // call a stored functor
22     bool c = pred1.call( 30 );
23 }
```

```
1 class Threshold25
2 {
3     public:
4         bool operator()( int x ) const
5         {
6             return ( x > 25 );
7         }
8 };
9
10 int main( void )
11 {
12     // create a functor
13     Threshold25 pred0();
14
15     // copy a functor
16     Threshold25 pred1 = pred0;
17
18     // call a stored functor
19     bool b = pred1( 15 );
20
21     // call a stored functor
22     bool c = pred1( 30 );
23 }
```

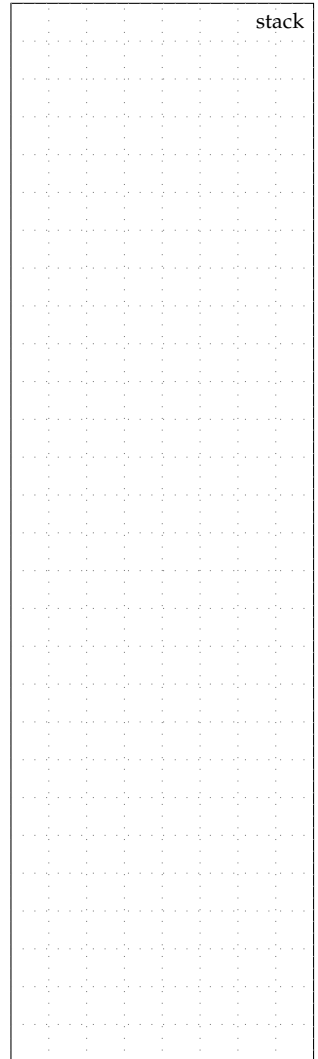
```
01 class Threshold25
02 {
03 public:
04     bool operator()( int x ) const
05     {
06         return ( x > 25 );
07     }
08 };
09
10 int main( void )
11 {
12     // create a functor
13     Threshold25 pred0();
14
15     // copy a functor
16     Threshold25 pred1 = pred0;
17
18     // call a stored functor
19     bool b = pred1( 15 );
20
21     // call a stored functor
22     bool c = pred1( 30 );
23 }
```

- Use arrow pointing to the *code* to represent a functor with no state



```
01 class Threshold
02 {
03     public:
04
05     Threshold( int t ) : m_t( t )
06     { }
07
08     bool operator()( int x ) const
09     {
10         return ( x > m_t );
11     }
12
13     private:
14     int m_t;
15 };
16
17 int main( void )
18 {
19     // create a functor
20     Threshold pred0(25);
21
22     // copy a functor
23     Threshold pred1 = pred0;
24
25     // call a stored functor
26     bool b = pred1( 15 );
27
28     // call a stored functor
29     bool c = pred1( 30 );
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

```
1  bool threshold_25( int x ) { return ( x > 25 ); }
2
3  class Threshold
4  {
5  public:
6      Threshold( int t ) : m_t( t ) { }
7      bool operator()( int x ) const { return ( x > m_t ); }
8  private:
9      int m_t;
10 };
11
12 template < typename S, typename Pred >
13 int count_if( const S& seq, Pred pred )
14 {
15     int count = 0;
16     for ( auto v : seq )
17         if ( pred(v) )
18             count++;
19     return count;
20 }
21
22 int main( void )
23 {
24     List<int> lst;
25     lst.push_front( 12 );
26     lst.push_front( 15 );
27     lst.push_front( 50 );
28     lst.push_front( 06 );
29     lst.push_front( 76 );
30
31     int a = count_if( lst, &threshold_25 );
32     int b = count_if( lst, Threshold(25) );
33     return 0;
34 }
```

```
01 class Threshold
02 {
03     public:
04
05     Threshold( int t ) : m_t( t )
06     { }
07
08     bool operator()( int x ) const
09     {
10         return ( x > m_t );
11     }
12
13     private:
14     int m_t;
15 };
16
17 template <>
18 int count_if<int[2],Threshold>(
19     const int[2]& seq,
20     Threshold    pred )
21 {
22     int count = 0;
23     for ( auto v : seq )
24         if ( pred(v) )
25             count++;
26     return count;
27 }
28
29 int main( void )
30 {
31     int arr[] = { 15, 35 };
32     int a = 25;
33     Threshold p(a);
34     int b = count_if( arr, p );
35     return 0;
36 }
```

stack

2. C++ Lambdas

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

```
1 int main( void )
2 {
3     int a = 25                // environment for functor
4
5     // creates an anonymous functor that explicitly captures a
6     auto pred0 = [a]( int x )
7     {
8         return x > a;
9     };
10
11    auto pred1 = pred0;       // copy a lambda
12    bool b = pred1( 15 );    // call a stored lambda
13    bool c = pred1( 30 );    // call a stored lambda
14 }
```

- 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

```
1 // lambda that implicitly captures a (by value)
2 auto pred0 = [=]( int x )
3 {
4     return x > a;
5 };
6
7 // lambda that implicitly captures a (by reference)
8 auto pred0 = [&]( int x )
9 {
10    return x > a;
11 };
```

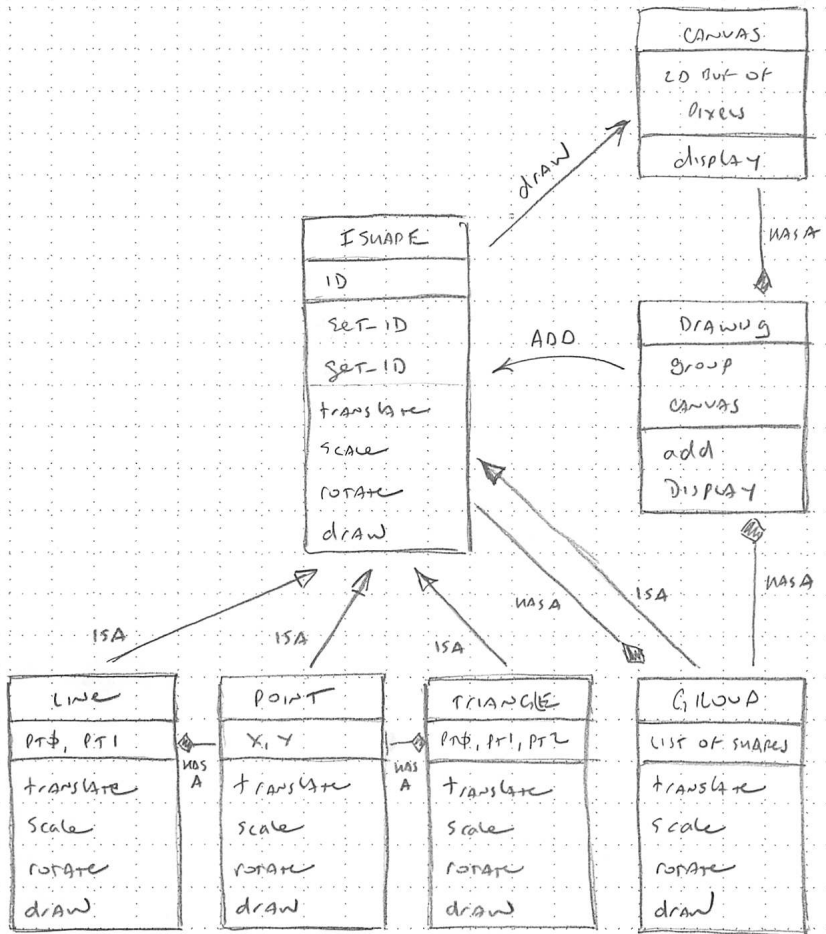
- Use templates to make algorithms generic over function pointers, functors, and lambdas

```
1  bool threshold_25( int x ) { return ( x > 25 ); }
2
3  class Threshold
4  {
5  public:
6      Threshold( int t ) : m_t( t ) { }
7      bool operator()( int x ) const { return ( x > m_t ); }
8  private:
9      int m_t;
10 };
11
12 template < typename S, typename Pred >
13 int count_if( const S& seq, Pred pred )
14 {
15     int count = 0;
16     for ( auto v : seq )
17         if ( pred(v) )
18             count++;
19     return count;
20 }
21
22 int main( void )
23 {
24     List<int> lst;
25     lst.push_front( 12 );
26     lst.push_front( 15 );
27     lst.push_front( 50 );
28     lst.push_front( 06 );
29     lst.push_front( 76 );
30
31     int a = count_if( lst, &threshold_25 );
32     int b = count_if( lst, Threshold(25) );
33     int c = count_if( lst, []( int x ) { return x > 25; } );
34     return 0;
35 }
```

```
01 template <>
02 int count_if<int[2],__lambda0>(
03     const int[2]& seq,
04     __lambda_0    pred )
05 {
06     int count = 0;
07     for ( auto v : seq )
08         if ( pred(v) )
09             count++;
10     return count;
11 }
12
13 int main( void )
14 {
15     int arr[] = { 15, 35 };
16
17     int a = 25;
18     auto p =
19         [=]( int v ) {
20             return v > a;
21         };
22
23     int b = count_if( arr, p );
24     return 0;
25 }
```

stack

3. Drawing Framework Case Study



- Create animations by repeatedly drawing and clearing screen

```
1 int main( void )
2 {
3     // Create a group of lines forming a star
4     Group star;
5     for ( int i = 0; i < 8; i++ )
6         star.add( Line( Point(0,0), Point(0,3) ) % (i*45) );
7
8     // Randomly place stars in a group
9     Group stars;
10    for ( int i = 0; i < 6; i++ ) {
11        int x_offset = ( rand() % 30 ) - 15;
12        int y_offset = ( rand() % 30 ) - 15;
13        stars.add( star + Point( x_offset, y_offset ) );
14    }
15
16    // Make it snow
17    for ( int i = 0; i < 80; i++ ) {
18
19        // Clear the screen
20        if ( i != 0 ) {
21            for ( size_t j = 0; j < 33; j++ )
22                printf("\x1b[A");
23        }
24
25        // Draw the snowflakes
26        Drawing drawing;
27        drawing.add( stars + Point(0,30-i) );
28        drawing.display();
29
30        // Wait between frames
31        usleep(100000);
32    }
33
34    return 0;
35 }
```

<https://repl.it/@cbatten/ece2400-T15-ex1>

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

```
1  template < typename DrawFrame >
2  void animate( int num_frames, DrawFrame draw_frame )
3  {
4      for ( int i = 0; i < num_frames; i++ ) {
5
6          // Clear the screen
7          if ( i != 0 ) {
8              for ( size_t j = 0; j < 33; j++ )
9                  printf("\x1b[A");
10         }
11
12         // Draw the frame
13         Drawing drawing;
14         draw_frame( i, &drawing );
15         drawing.display();
16
17         // Wait between frames
18         usleep(100000);
19     }
20 }
21
22 int main( void )
23 {
24     ...
25
26     animate( 80, [&]( int i, Drawing* drawing_p ) {
27         drawing_p->add( stars + Point(0,30-i) );
28     });
29
30     return 0;
31 }
```

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