ECE 2400 Computer Systems Programming
Fall 2018

Topic 14: Functional Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2018-11-19-00-11

1 C++ Functors

2 C++ Lambdas
• 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 ADT (seq), a value to search for, and returns the number of elements in the sequence that match the given value as an \(\tau\)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 \(\text{List}<\text{int}>\), a \(\text{List}<\text{float}>\), a \(\text{Vector}<\text{int}>\), etc. \textit{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

```c
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) ) // notice dereference is optional!
            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 = pred0; // 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 function pointers and 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;
}
```
class Threshold
{
    public:
    Threshold( int threshold )
        : m_threshold( threshold )
    { }

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

    private:
    int m_threshold;
};

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

int main( void )
{
    int arr[] = { 15, 35 };
    int x = 25;
    auto p = Threshold(x);
    int c = 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 x = 25  // environment for functor

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

    auto pred1 = pred0;  // copy a lambda
    bool a = pred1( 15 );  // call a stored lambda
    bool b = 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 x (by value)
auto pred0 = [=]( int v )
{
    return v > x;
};

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

```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) );
    int c = count_if( list, []( int v ) { return v > 25; } );
    return 0;
}
```
```cpp
#include <iostream>
#include <array>

// A lambda function to check if an integer is greater than 25.

// A lambda function that counts elements in an array that are greater than 25.

int main() {
    int arr[] = { 15, 35 }; // An array of integers.
    int x = 25;             // An integer.
    auto p = [] ( int v ) { // A lambda function.
        return v > 25;      // Returns true if v is greater than 25.
    };

    int c = count_if( arr, p ); // Count elements in arr that are greater than 25.
    return 0;                    // Return 0.
}
```

```