ECE 2400 Computer Systems Programming
Fall 2020

Topic 15: Functional Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2020-11-30-01-25

1 C Function Pointers 3

2 C++ Functors 7

3 C++ Lambdas 11
• 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

1. C Function Pointers

• Code is also stored in memory, so a *function pointer* points to code
• Enables passing a *function* as a parameter to a different function
• Consider the following functions:

```c
1 int lt( int x, int y ) { return x < y; }
2 int gt( int x, int y ) { return x > y; }
```

• The type of a function is the function’s *signature*
• A function signature includes the parameter types and return type
• We don’t really care about the function or parameter names
• `lt` and `gt` have the same function signature and thus the same type
• `lt` and `gt` are essentially two “values” of the same type
• We can write the type of these functions as follows:

```c
1 int ( int, int )
```
• Recall that every type T has a corresponding pointer type T *
• So the type of a pointer to this function might look like this:

\[
\text{(int ( int, int ))*}
\]

• ... and declaring a variable to hold a function pointer like this:

\[
\text{(int ( int, int ))* cmp_ptr;}
\]

• This makes sense and would be consistent, but C actually uses a slightly different syntax for declaring a function pointer:

\[
\text{int (*cmp_ptr) ( int, int );}
\]

• This also makes sense since now there is a direct connection between a function declaration and a function pointer:

\[
\text{int lt ( int x, int y );}
\]
\[
\text{int (*cmp_ptr) ( int, int );}
\]

• The type of a function pointer is complex, so use a typedef

\[
\text{typedef int (*cmp_ptr_t) ( int, int );}
\]
\[
\text{cmp_ptr_t cmp_ptr;}
\]

• The address-of operator (&) applied to a function name evaluates to a pointer to that function

\[
\text{typedef int (*cmp_ptr_t) ( int, int );}
\]
\[
\text{cmp_ptr_t cmp_ptr = &gt;}
\]

• We can dereference a function pointer and use the call operator (()) to call a function via function pointer

\[
\text{typedef int (*cmp_ptr_t) ( int, int );}
\]
\[
\text{cmp_ptr_t cmp_ptr = &gt;}
\]
\[
\text{int result = (*cmp_ptr)( 3, 4 );}
\]
#include <stdio.h>

int lt( int x, int y ) { return x < y; }
int gt( int x, int y ) { return x > y; }

typedef int (*cmp_ptr_t) ( int, int );

void sort( int* x_ptr, int* y_ptr, cmp_ptr_t cmp_ptr )
{
    // Dereference function pointer, then call function
    if ( ((*cmp_ptr)(*x_ptr), (*y_ptr))) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}

void sort3( int* x_ptr, int* y_ptr, int* z_ptr,
            cmp_ptr_t cmp_ptr )
{
    sort( x_ptr, y_ptr, cmp_ptr );
    sort( y_ptr, z_ptr, cmp_ptr );
    sort( x_ptr, y_ptr, cmp_ptr );
}

int main( void )
{
    int a = 9; int b = 5; int c = 3;
    sort3( &a, &b, &c, &gt );
    printf(" %d < %d < %d \n", a, b, c );
    sort3( &a, &b, &c, &lt );
    printf(" %d > %d > %d \n", a, b, c );
    return 0;
}
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;
}
```
2. 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 x ) const
  {
    return ( x > m_threshold );
  }

private:

  int m_threshold;
};

int main( void )
{
  int a = 25 // environment for functor
  Threshold pred0(a); // explicit closure
  Threshold pred1 = pred0; // copy a functor
  bool b = pred1.call( 15 ); // call a stored functor
  bool c = pred1.call( 30 ); // call a stored functor
}
```
2. C++ Functors

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

```cpp
class Threshold
{

public:

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

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

private:
    int m_threshold;
};

int main(void)
{
    int a = 25; // environment for functor
    Threshold pred0(a); // explicit closure
    Threshold pred1 = pred1; // copy a functor
    bool b = pred1(15); // call a stored functor
    bool c = pred1(30); // call a stored functor
}
```
- 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 threshold) : m_threshold(threshold) { }
    bool operator()(int x) const { return (x > m_threshold); }
private:
    int m_threshold;
};

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 threshold)
        : m_threshold(threshold)
    {
    }

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

    private:
    int m_threshold;
};

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;
}
3. 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 threshold ) : m_threshold( threshold ) { }
    bool operator()( int x ) const { return ( x > m_threshold ); }
  private:
    int m_threshold;
};

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;
}
```
```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;
}
```