CS 247: Software Engineering Principles

C++ lambdas, bind, mem_fn
Recap: Can Specialize STL Algorithms

```cpp
void quote( string word ) {
    cout << "\"" << word << "\"" << endl;
}

int main() {
    vector<string> words;

    // apply quote to each string in the vector words
    for_each ( words.begin(), words.end(), quote );
}
```
Recap: Can Specialize STL Algorithms

- STL algorithms can be specialized with user-provided functions and functors.
- STL algorithms are elegant, but only because of all supporting code (functions, functors, functor adapters).
- Such functions and functors are declared like official functions and classes.
  - Pollutes the global namespace.
C++11 Facilities to Simplify Supporting Code

• **lambdas** – to ease the task of writing functions and functors

• **bind** – unifies a number of function adaptors from C++03

• **mem_fn** – unifies the function adaptors `mem_fun` and `mem_fun_ref` for accessing member functions of the iterator referent
C++ Lambdas

A **lambda expression** is like an unnamed, inline function that can be used wherever a callable object is expected, without having to pre-define it.

```cpp
for_each ( begin(words), end(words),
    [](string s) -> void { cout << "\\n" << s << "\\n" << endl; });
```
Syntax

[ capture list ] ( parm list ) → return type { function body }

- **Capture list** – (often empty) lists local variables used within the function
- **Parameter list** – usual function parameter list
- **Return type** – may be inferred from the return statement(s) in the function body. Specify if need to convert types, known as *trailing return type*.
- **Function body** – normal function body

The capture list and parameter list can be empty, and the return type inferred, so the following is a valid lambda:

```
[] { cout << "Hello, world! << endl; }
```
C++ Lambdas

A lambda expression creates a thing that can be saved and treated like a function pointer or function object.

- It can be called right away:

```cpp
[] (string s) { cout << "\"" << s << "\"" << endl; } ("yak");
```

- Or it can be stored in a variable, and the variable is used as a function pointer or functor:

```cpp
auto quote = [](string s){ cout << "\"" << s << "\"" << endl; }; quote ("yak!");
```

- Mostly, lambdas are “bare” unnamed declarations used “in place” where a callable function is expected.
Capture List

A lambda can be used to create an unnamed, ad-hoc functor, initialized by the capture list.

- A function object that holds captured values is called a closure.

```cpp
string colour;
cin >> colour;
cout << count_if( begin(list), end(list),
    [ colour ] (Balloon* b) { return b->colour() == colour; } ) <<
    colour << " balloons." << endl;

// compiler effectively translates the above lambda into a functor
class the_lambda {
public:
    the_lambda ( string colour_ ) : colour(colour_) {}
    bool operator() ( Balloon* b) const { return b->colour == colour; }
private:
    string colour;
};
```
Capture List

The capture list refers to the value of local variables at the point of the lambda's creation

cout << "Enter balloon colour of interest: "; // = red
cin >> colour;
auto count = [ colour ] (Balloon* b) { return b->colour() == colour; };
// count functor looks for red balloons

cout << "There are " << count_if ( list.begin(), list.end(), count ) << " balloons." << endl; // looks for red balloons

cout << "Enter balloon colour of interest: "; // = green
cin >> colour;
cout << "There are " << count_if ( list.begin(), list.end(), count ) << " balloons." << endl; // looks for red balloons

cout << "Enter balloon colour of interest: "; // = violet
cin >> colour;
cout << "There are " << count_if ( list.begin(), list.end(), count ) << " balloons." << endl; // looks for red balloons
Capture by Reference

BUT you can specify capture by reference, which means that the lambda holds a reference to the actual variable

cout << "Enter balloon colour of interest: "; // = red
cin >> colour;
auto count = [&colour] (Balloon* b) { return b->colour() == colour; };
// count functor looks for red balloons

cout << "Enter balloon colour of interest: "; // = green
cin >> colour;
cout << "There are " << count_if ( begin(list), end(list), count ) << " balloons."
   << endl; // looks for green balloons

cout << "Enter balloon colour of interest: "; // = violet
cin >> colour;
cout << "There are " << count_if ( begin(list), end(list), count ) << " balloons."
   << endl; // looks for violet balloons
Especially Useful with STL Algorithms

class Balloon {
public:
    Balloon ( string colour, int volume ) : colour_(colour), volume_(volume) {} 
    string colour() const { return colour_; }  
    int volume() const { return volume_; }  
    void shrink( int amount ) { volume_ -= amount; }  
    void pretty_print ( ostream& sout ) {  
        sout << colour() << " balloon of size " << volume() << "cm^3" << endl; }  
private:
    string colour_; 
    int volume_; 
};

// a custom function object class
class PrettyPrint {
public:
    PrettyPrint(ostream& output_stream_) :output_stream(output_stream_) {} 
    void operator() (Balloon* ptr) const { ptr->pretty_print(output_stream); }  
private:
    ostream& output_stream; 
};

int main () {
    ofstream out_file("output.txt"); // open a file for output
    ... 
    // elegant code using STL algorithm
    for_each(list.begin(), list.end(), PrettyPrint(out_file)); 
    ... 
}
Especially Useful with STL Algorithms

class Balloon {
  public:
    Balloon ( string colour, int volume ) : colour_{colour}, volume_{volume} {}  
    string colour() const { return colour_; }  
    int volume() const { return volume_; }  
    void shrink( int amount ) { volume_ -= amount; }  
    void pretty_print ( ostream& sout ) {  
      sout << colour() << " balloon of size " << volume() << " cm^3 " << endl; }

  private:
    string colour_;  
    int volume_;  
};

int main () {  
  ofstream out_file("output.txt"); // open a file for output  
  ...

  // elegant code using STL algorithm  
  for_each( begin(list), end(list),  
            [&out_file](Balloon * b){b->pretty_print(out_file);} );

  for ( auto it : list ) delete it;
}
Some Gotchas!

• A captured reference is only valid if the referred-to variable still exists!

• By default, a lambda may not change the value of a captured variable that it copies by value.
  — To modify a captured variable, lambda must be mutable.
  
  [ capture list ] ( parm list ) mutable -> return …

• Using lambdas inside member functions – data members are not local variables, and thus cannot be captured!
  – They are members of object *this.
Example

class Collection {
public:

...  
  int numAboveThreshold() const {
    int result = 0;
    for_each( begin(ints), end(ints),
        [&_threshold,&result](int i) mutable {if (i>_threshold) result++;} );
    return result;
  }

private:
  vector<int> ints;
  int _threshold;
};

int main() {
  ...
  cout << "There are " << c.numAboveThreshold() <<
      " elements above " << c.thresholdIs() << "." << endl;
}
This Code Works (C++11)

class Collection {
public:
    ...
    int numAboveThreshold() const {
        int result = 0;
        for_each( begin(ints), end(ints),
                    [this,&result](int i) mutable { if (i > _threshold) result++; } );
        return result;
    }
private:
    vector<int> ints;
    int _threshold;
};

int main() {
    ...
    cout << "There are " << c.numAboveThreshold() <<
         " elements above " << c.thresholdIs() << "." << endl;
}
class Collection {
public:
...
int numAboveThreshold() const {
    int result = 0;
    for_each(begin(ints), end(ints),
        [_threshold = _threshold,&result](int i) mutable {
            if (i > _threshold) result++;
        });
    return result;
}
private:
    vector<int> ints;
    int _threshold;
};

int main() {
...
    cout << "There are " << c.numAboveThreshold() << " elements above " << c.thresholdIs() << "." << endl;
C++11’s bind

C++11’s bind replaces a number of adapters and binders from C++98:

- ptr_fun, mem_fun, mem_fun_ref, bind1st, bind2nd

C++ Libraries

#include <functional>
using namespace std;
using namespace std::placeholders; // used for _1, _2, etc.
C++11’s bind

**bind** is a functor adapter used to associate function parameters with particular arguments or values.

– Usually to reduce the number of arguments that need to be provided when the function/functor is called.

```cpp
int subtract ( int a, int b ) {
    return a - b;
}

// assume i and i2 are int variables
cout << bind( subtract, i, i2)() << endl;

is equivalent to

cout << subtract (i, i2) << endl;
```
bind ( function pointer, bound args ) ( call args )

• function pointer

• bound args – a list of argument values that are bound to function parameters.
  – first bound argument corresponds to first function argument
  – second bound argument corresponds to second function argument, etc.
  – bound argument might be a fixed value, or value might be deferred until the function is called

• call args – argument values that are provided when the function is called
How it Works

Can think of `bind` as returning a function object that:

- stores a pointer to the function as a data member,
- copies the bound arguments and stores them as data members, and
- overloads `operator()` to call the saved function pointer, passing the saved argument values as function arguments.
Call by Reference

• Bound arguments are copied to and stored in function object.

• If bound to a call-by-reference argument in the function, the functor’s internal copy is modified (but not the original program variable).

```c
void mod23 ( int x, int& y, int& z ) {
    y = x + y;
    z = y + z;
}

... bind (mod23, i1, i2, i3)();
// has no effect on variables i1, i2 or i3
```
Reference Wrapper Class

Can modify the bound program variable by wrapping the argument in a reference wrapper class.

- Has the effect of a copyable reference!

```c
void mod23 ( int x, int& y, int& z ) {
    y = x + y;
    z = y + z;
}
bind(mod23, i1, i2, ref(i3))();  // i1 and i2 are unaffected;
    // i3 is modified
```
Placeholders

- Bind is most useful when binding a subset of function arguments, allowing others to be set by call arguments.
- Use placeholders to indicate which call argument maps to a particular function parameter.

```c
void inc ( int &x, int amount ) { x += amount; }

int main() {
    int i = 52;
    bind ( inc, ref(i), 100 ) ();
    bind ( inc, ref(i), _1 ) ( 100 );
    bind ( inc, _1, 100 ) ( i );
    bind ( inc, _2, _1 ) ( 100, i );
    auto bi = bind ( inc, ref(i), _1 );
    bi(3);
    bi(10);
}
```
Call by Reference

void inc ( int &x, int amount ) { x += amount; }

• if bound argument is a wrapped reference to a modifiable location (an lvalue), the function can modify it
  
  bind ( inc, ref(i), 100 ) ();
  bind ( inc, ref(i), i2 ) ();

• if call argument maps to reference parameter, the function can modify it
  
  bind ( inc, _1, 100 ) ( i );
  bind ( inc, _2, _1 ) ( 100, i);

• if the argument is not a modifiable location (e.g. constant, literal, expression, function call), get a compiler error only if try to change the location!
  
  bind ( inc, 5, 100) ();
  bind ( inc, _1, 100 ) ( 5 );
STL Examples

```cpp
void inc ( int &x, int amount ) { x += amount; };
int main() {
    int ints[] = {10,20,30,40,50,60,70};
    for_each ( begin(ints), end(ints), bind(inc, _1, 100) );

    transform ( begin(ints), end(ints), begin(ints),
            bind(plus<int>(), _1, 5) );
}

void mod23 ( int x, int& y, int& z ) {
    y = x + y;
    z = y + z;
}
int main() {
    vector<int> ones (10,1);
    for_each ( begin(ones), end( ones ), bind( mod23, 5, 3, _1) );
```
Binding Objects and Member Functions

class Simple {
public:
    Simple (int i) : val_{i} {}  
void print() const { cout << "Simple " << val_ << endl; }  
void write_to( ostream& sout ) const {
        sout << "Simple " << val_ << endl; }
    int val() const { return val_; }  
void valIs ( int i ) { val_ = i; }
void inc () { val_++; }
private:
    int val_; 
};

// Non member functions as well
void print_Simple( Simple s ) { s.print(); }
void inc_Simple ( Simple& s ) { s.inc(); }
void update_Simple ( Simple& s, int val ) { s.valIs(val); }
Non-member functions are no different from the ordinary functions discussed so far.

Simple s{1};
bind (inc_Simple, s)();  // modifies a copy of s
bind (print_Simple, s)();
bind (inc_Simple, ref(s))();  // modifies s
bind (print_Simple, s)();
bind (inc_Simple, _1) (s);  // modifies s
bind (print_Simple, _1) (s);
Binding of Member Functions

When binding a member function

– must provide a pointer to the member function
– must provide this argument (the object whose member function is invoked must be the first argument)

Simple s{1};

bind( &Simple::inc, s )(); // modifies copy of s
bind( &Simple::valIs, s, 10)();
bind( &Simple::print, s )();

bind( &Simple::inc, ref(s) )(); // modifies s
bind( &Simple::valIs, ref(s), ref(s), 10)();
bind( &Simple::print, s )();

bind ( &Simple::inc, _1 ) (s);
bind ( &Simple::valIs, _1, _2 ) (s, 100); // modifies s
bind ( &Simple::print, _1 ) (s);
Applied to Containers of Objects

As simple to apply to containers of objects as to apply to containers of integers.

```cpp
vector<Simple> simps (10, Simple{1});

for_each( begin(simps), end(simps), bind( &Simple::inc, _1) );

for_each ( cbegin(simps), cend(simps), bind( &Simple::print, _1) );

for_each ( begin(simps), end(simps), bind (&Simple::valIs, _1, 10) );

for_each ( cbegin(simps), cend(simps),
            bind (&Simple::write_to, _1, ref(cout)) );
```
C++11 also provides `mem_fn` that adapts a function pointer for an object’s member function.

```cpp
for_each ( begin(list), end(list), mem_fn( &Balloon::print ) );
```

- Easier to use than `bind` if there are no arguments to bind.
- Works for collections of objects and pointers to objects.
- Subsumes `mem_fun` and `mem_fun_ref` from C++03.