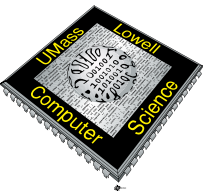
Lambda Expressions Part 2

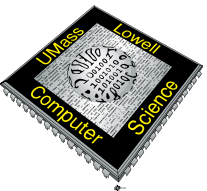
COMP.2040 – Computing IV

Dr. Tom Wilkes



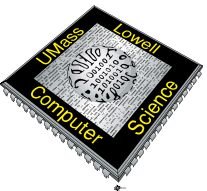
Sources

- Scott Meyers, *Effective Modern C++*, Chapter 6
- Bjarne Stroustrup, *The C++ Programming Language* (4th edition), Section 11.4
- Bjarne Stroustrup, *Programming: Principles and Practice Using C++* (2nd edition)



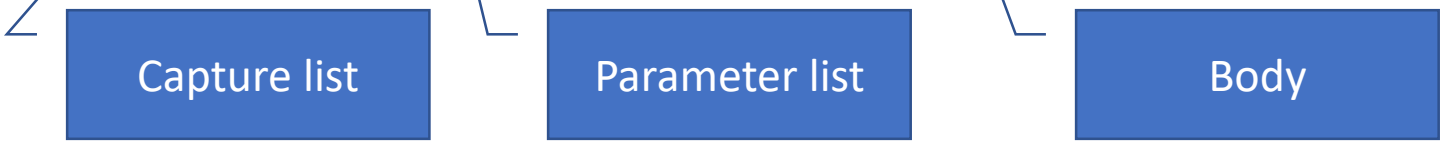
Example: `print_modulo` implemented using a local class

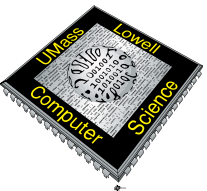
```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    class Modulo_print {
        ostream& os; // members to hold the capture list
        int m;
    public:
        Modulo_print (ostream& s, int mm): os(s), m(mm) {}
        void operator()(int x) const
            { if (x%m==0) os << x << '\n'; }
    };

    for_each( begin(v), end(v), Modulo_print{os,m} );
}
```



Example: `print_modulo` implemented using a lambda expression

```
void print_modulo(const vector<int>& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    for_each(begin(v),end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; });
}
```

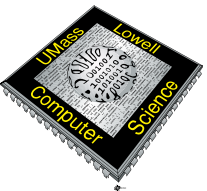




Syntax of a lambda expression

A lambda expression consists of a sequence of parts:

- A possibly empty *capture list*, specifying what names from the definition environment can be used in the lambda expression's body, and whether those are copied or accessed by reference. The capture list is delimited by **[]**.
- An optional *parameter list*, specifying what arguments the lambda expression requires. The parameter list is delimited by **()**.
- An optional **mutable** specifier, indicating that the lambda expression's body may modify the state of the lambda (i.e., change the lambda's copies of variables captured by value).
- An optional **noexcept** specifier.
- An optional *return type declaration* of the form **–>** *type*.
- A *body*, specifying the code to be executed. The body is delimited by **{}**.

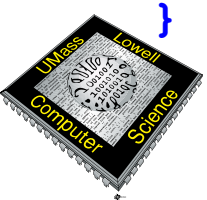


Example: `print_modulo` implemented using a for-loop

```
template<class C>
void print_modulo(const C& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    for (auto x : v)
        if (x%m==0) os << x << '\n';
}
```

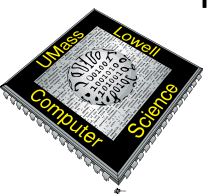
The for-loop implementation above does depth-first traversal, but cannot be changed to do breadth-first. For that we need an algorithm!

```
template<class C>
void print_modulo(const C& v, ostream& os, int m)
    // output v[i] to os if v[i]%m==0
{
    breadth_first( begin(v), end(v),
        [&os,m](int x) { if (x%m==0) os << x << '\n'; } );
}
```



Capture list (“lambda introducer”) options

- **[]**: an empty capture list. This implies that no local names from the surrounding context can be used in the lambda body. For such lambda expressions, data is obtained from arguments or from nonlocal variables.
- **[&]**: implicitly capture by reference. All local names can be used. All local variables are accessed by reference.
- **[=]**: implicitly capture by value. All local names can be used. All names refer to copies of the local variables taken at the point of call of the lambda expression.
- **[*capture-list*]**: explicit capture; the *capture-list* is the list of names of local variables to be captured (i.e., stored in the object) by reference or by value. Variables with names preceded by **&** are captured by reference. Other variables are captured by value. A capture list can also contain **this** and names followed by **...** as elements.
- **[&, *capture-list*]**: implicitly capture by reference all local variables with names not mentioned in the list. The capture list can contain **this**. Listed names cannot be preceded by **&**. Variables named in the capture list are captured by value.
- **[=, *capture-list*]**: implicitly capture by value all local variables with names not mentioned in the list. The capture list cannot contain **this**. The listed names must be preceded by **&**. Variables named in the capture list are captured by reference.

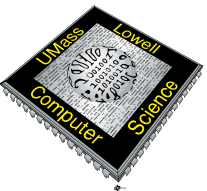


Mutable lambdas

Usually, we don't want to modify the state of the function object (the closure), so by default we can't. That is, the **operator()()** for the generated function object is a **const** member function. In the unlikely event that we want to modify the state (as opposed to modifying the state of some variable captured by reference), we can declare the lambda **mutable**. For example:

```
void algo(vector<int>& v)
{
   int count = v.size();
   std::generate( v.begin(),v.end(),
      [count]() mutable { return --count; } );
}
```

The **--count** decrements the copy of **v**'s size stored in the closure.



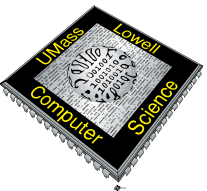
Call and Return—1

The rules for passing arguments to a lambda are the same as for a function, and so are the rules for returning results.

In fact, with the exception of the rules for capture most rules for lambdas are borrowed from the rules for functions and classes.

However, two irregularities should be noted:

1. If a lambda expression does not take any arguments, the argument list can be omitted. Thus, the minimal lambda expression is **[]{}**.
2. A lambda expression's return type can be deduced from its body. Unfortunately, that is not also done for a function.



Call and Return—2

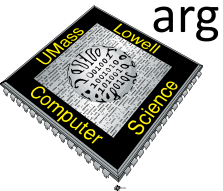
If a lambda body does not have a **return**-statement, the lambda's return type is **void**.

If a lambda body consists of just a single **return**-statement, the lambda's return type is the type of the **return**'s expression.

If neither is the case, we have to explicitly supply a return type. For example:

```
void g(double y)
{
    [&]{ f(y); } // return type is void
    auto z1 = [=](int x){ return x+y; } // return type is double
    auto z2 = [=,y]{ if (y) return 1; else return 2; }
       // error: body too complicated for return type deduction
    auto z3 =[y]() { return 1 : 2; } // return type is int
    auto z4 = [=,y]() -> int { if (y) return 1; else return 2; }
       // OK: explicit return type
}
```

When the suffix return type notation is used, we cannot omit the argument list.



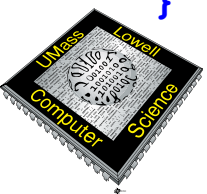
The Type of a Lambda

Cannot use an auto variable before its type has been deduced:

```
auto rev = [&rev](char* b, char* e)
    { if (1 < e-b) { swap(*b,*--e); rev(++b,e); } };
// error
```

Can solve this by introducing a name and then using it:

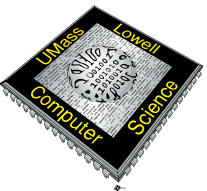
```
void f(string& s1, string& s2)
{
    function<void(char* b, char* e)> rev =
        [&](char* b, char* e)
            { if (1 < e-b) { swap(*b,*--e); rev(++b,e); } };

    rev( &s1[0], &s1[0]+s1.size() );
    rev( &s2[0], &s2[0]+s2.size() );
}
```



A Lambda Expression as a Callback—1

In a sample graphics library, for each action on a Widget, we have to define two functions: one to map from the system's notion of a callback and one to do our desired action. Consider:

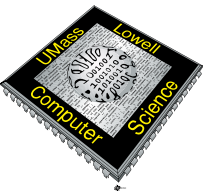
```
struct Simple_window : Graph_lib::Window
{
    Simple_window{Point xy, int w, int h, const string& title};
    void wait_for_button(); // simple event loop

private:
    Button next_button; // the "Next" button
    bool button_pushed; // implementation detail
    static void cb_next(Address, Address);
        // callback for next_button
    void next();
        // action to be done when next_button is pressed
};
```



A Lambda Expression as a Callback—2

By using a lambda expression, we can eliminate the need to explicitly declare the mapping function cb_next(). Instead, we define the mapping in Simple_window's constructor:

```
Simple_window::Simple_window(Point xy, int w, int h,
                             const string& title)
    : Window{ xy, w, h, title },
      next_button{ Point{x_max()-70,0}, 70, 20,"Next",
          [](Address, Address pw)
           { reference_to<Simple_window>(pw).next(); }
      },
      button_pushed{ false }
{
    attach(next_button);
}
```



[For full code example, see Chapter 16 of:
Stroustrup, *Programming: Principles and Practice Using C++* (2nd edition)]