

CS100

Introduction to Programming

Lecture 25. Lambda functions
(Or: “New” features of C++11/14)

auto Variables

- Implicit variable declaration: the compiler knows what you mean
 - (Almost) necessary for anonymous types
 - Very convenient for complex templates
 - Easily abused by lazy programmers!

```
std::map<int, std::list<string>> M;  
auto iter = M.begin(); //what's the type of iter?  
auto pair = std::make_pair(iter, M.key_range(...));  
auto ptr = condition ? new class1 : new class2; //ERROR  
auto x = 15; auto s = (string)"Hello";
```

Taking It Further...

- Some people now use auto *exclusively*
 - Beauty is in the eyes of the beholder...

```
auto flags = DWORD { FILE_FLAG_NO_BUFFERING };  
auto numbers = std::vector<int> { 1, 2, 3 };  
auto event_handle = HANDLE {};
```

// You can consistently use ,auto' with functions, too:

```
auto is_prime(unsigned n) -> bool;  
auto read_async(HANDLE h, unsigned count) -> vector<char>;
```

Today's learning objectives

- Functional programming
- Introduction to Lambda functions
- Playing with Lambda functions
- "New" features of C++11
- What's new in C++ 14?

Today's learning objectives

- Functional programming
- Introduction to Lambda functions
- Playing with Lambda functions
- "New" features of C++11
- What's new in C++ 14?

Two types of programming paradigms

- Imperative programming
 - Based on *von Neumann* architecture
 - Efficiency is primary concern, rather than suitability of language for software development
- Functional programming
 - Based on mathematical functions
 - Solid theoretical basis close to user
 - Unconcerned with architecture of machines

Example: Imperative programming

- Statements, not just expressions
 - Change the state of the code

- Examples

- Assignment:

```
int y;  
y = someFunction(x) ;
```

- Non-const member function

```
SomeClass class;  
class.someMethod(x) ;
```

Call is changing
internal state of
class

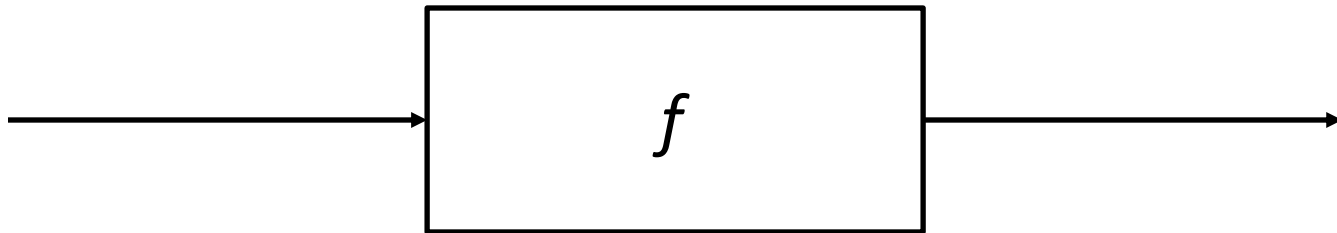
Function without
return value!

Imperative programming

- Drawbacks:
 - Depends on local/global state
 - May produce different result everytime a function is run
 - No transparency
 - Difficult to understand/predict the behavior of code

Functional programming

- Declarative programming paradigm
 - Uses expressions or declarations instead of statements
 - Avoids changing state/mutable data
 - Function output depends only on passed input variables
 - Function result is always the same



Origins: mathematical functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

$$\lambda (x) \quad x * x * x$$

for the function $\text{cube } (x) = x * x * x$

Lambda expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

e.g., $(\lambda (x) \ x * x * x) (2)$

which evaluates to 8

Functional forms

- A higher-order function, or *functional form*, is one that
 - takes functions as parameters
 - yields a function as its result
 - Both
- Example: Functional composition
 - Takes two functions as parameters
 - Yields function whose value is first actual parameter function applied to the application of the second
 - Form: $h \equiv f \circ g$, means $h(x) \equiv f(g(x))$

For $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$,
 $h \equiv f \circ g$ yields $(3 * x) + 2$

Functional Programming is really out There

- Functional Programming eXchange
- Strange Loop
- ReactConf
- LambdaConf
- LambdaJam
- CraftConf
- MSFT MVP Summit
- Qcon NYC/SF/London
- Closure West
- Spring into Scala
- Progressive F#
- FP Days
- SkillsMatter



Today's learning objectives

- Functional programming
- Introduction to Lambda functions
- Playing with Lambda functions
- "New" features of C++11
- What's new in C++ 14?

Lambda Functions

```
int main() {  
    [](){}();  
    []{}();  
} //this is legal C++,  
    //although not useful
```

Lambda Cometh to C++

- Lambda Expressions
 - `expr.prim.lambda`
- Anonymous functions

```
void absort(float* x, unsigned N) {  
    std::sort( x, x + N,  
        [](float a, float b) {  
            return std::abs(a) < std::abs(b) ;  
        }) ;  
}
```


Good Old C++

```
class Comp {  
    float a;  
public:  
    Comp(float x) {  
        a = x;  
    }  
  
    bool compare(float b) const {  
        return std::abs(a) < std::abs(b);  
    }  
};  
  
float array[5] = { 0, 1, 2, 3, 4 };  
float a = 3;  
Comp f(a);  
for(float item : array)  
    std::cout << std::boolalpha << f.compare(item) ;
```

Still, Good Old C++

```
class Comp {  
    float a;  
public:  
    Comp(float x) {  
        a = x;  
    }  
  
    bool operator () (float b) const {  
        return std::abs(a) < std::abs(b);  
    }  
};  
  
float array[5] = { 0, 1, 2, 3, 4 };  
float a = 3;  
Comp f(a);  
for(float item : array)  
    std::cout << std::boolalpha << f(item);
```

Not Valid C++

```
class ##### {  
    float a;  
public:  
    Foo(float x) {  
        a = x;  
    }  
  
    bool operator () (float b) const {  
        return std::abs(a) < std::abs(b);  
    }  
};  
  
float array[5] = { 0, 1, 2, 3, 4 };  
float a = 3;  
auto f = #####(a);  
for(float item : array)  
    std::cout << std::boolalpha << f(item);
```

In C++, Lambda are just function objects

```
class ##### {  
    float a;  
public:  
    Foo(float x) {  
        a = x;  
    }  
    bool operator () (float b) const {  
        return std::abs(a) < std::abs(b);  
    }  
};
```

```
float array[5] = { 0, 1, 2, 3, 4 };
```

```
float a = 3;
```

```
auto f = #####(a);
```

```
auto f = [a](float b) {return std::abs(a) < std::abs(b)};
```

```
for(float item : array)
```

```
    std::cout << std::boolalpha << f(item);
```

Lambdas in C++11/14

- Anonymous functions
- Written exactly in the place where it's needed
- Can access the variables available in the enclosing scope (closure)
- May maintain state (mutable or const)
- Can be passed to a function
- Can be returned from a function
- Deduce return type automatically
- Accept generic parameter types (only in C++14)

```
[a] (auto b) { return std::abs(a) < std::abs(b) } ;
```

Write Lambda function

- Write an Identity function that takes an argument and returns the same argument.

```
Identity(3)    //3
```

```
auto Identity = [] (auto x) {  
    return x;  
};
```

Write Lambda function

- Write 3 functions **add**, **sub**, and **mul** that take 2 parameters each and return their sum, difference, and product respectively.

```
add(3, 4) //7
```

```
sub(4, 3) //1
```

```
mul(4, 5) //20
```

Write Lambda function

```
auto add = [] (auto x, auto y) {  
    return x + y;  
};  
auto sub = [] (auto x, auto y) {  
    return x - y;  
};  
int mul (int x, int y) {  
    return x * y;  
};
```


Capturing

- Capturing external local variables
 - Compiled to a function object with fields

```
int fib1 = 1, fib2 = 1;
auto next_step = [&]() {           //capture by reference
    int temp = fib2; fib2 = fib2 + fib1; fib1 = temp;
};
for (int i = 0; i < 20; ++i) next_step();

std::thread t([=]() {             // capture by value
    std::cout << fib1 << std::endl;
});
t.join();
```

Write Lambda function

- Write a function *identityf* that takes an argument and returns a callable that returns that argument.

```
auto idf = identityf(5);  
idf() //5
```

Using an Inner Class

```
auto identityf = [] (auto x) {  
    class Inner {  
        int x;  
        public: Inner(int i): x(i) {}  
        int operator() () { return x; }  
    };  
    return Inner(x);  
};  
auto idf = identityf(5);  
idf() //5
```

Using a Closure

```
auto identityf = [] (auto x) {  
    return [] () { /* must remember x */ };  
};
```

```
auto identityf = [] (auto x) {  
    return [=] () { return x; };  
};
```

```
auto idf = identityf(5);  
idf() //5
```

A closure is a lambda expression paired with an environment that binds each of its free variables to a value

Closure != Lambda

- A lambda is just an anonymous function.
- A closure is a function which closes over the environment in which it was defined.
- Not all closures are lambdas and not all lambdas are closures.
- Closures are just function objects in C++
- C++ closures **do not extend the lifetime of their context** (If you need this use `shared_ptr`)

Today's learning objectives

- Functional programming
- Introduction to Lambda functions
- **Playing with Lambda functions**
- "New" features of C++11
- What's new in C++ 14?

Write Lambda function

- Write a function that produces a function that returns values in a range.

`fromto(0, 10)`

Write Lambda function

```
auto fromto = [](auto start, auto finish) {  
    return [=] () mutable {  
        if (start < finish)  
            return start++;  
        else  
            throw std::runtime_error("Complete");  
    };  
};
```

mutable: state can be changed!!

```
auto range = fromto(0, 10)  
range() //0  
range() //1
```


Write Lambda function

- Write a function that adds from two invocations

```
addf (5) (4) //9
```

```
auto addf = [] (auto x) {  
    return [=] (auto y) {  
        return x+y;  
    };  
};
```

Write Lambda function

- Write a function swap that swaps the arguments of a binary function.

```
swap(sub) (3, 2) // -1
```

Write Lambda function

```
auto sub = [] (auto x, auto y) {  
    return x - y;  
};
```

```
auto swap = [] (auto binary) {  
    return [=] (auto x, auto y) {  
        return binary(y, x);  
    };  
};
```

```
swap(sub) (3, 2) //-1
```

Write Lambda function

- Write a function `twice` that takes a binary function and returns a unary function that passes its argument to the binary function twice.

```
twice (add) (11) // 22
```

Write Lambda function

```
auto twice =[] (auto binary) {  
    return [=] (auto x) {  
        return binary(x, x);  
    };  
};
```

```
twice(add) (11) // 22
```

Write Lambda function

- Write a function that takes a binary function and makes it callable with two invocations.

```
applyf (mul) (3) (4) // 12
```

Write Lambda function

```
auto applyf = [] (auto binary) {  
    return [binary] (auto x) {  
        return [binary,x] (auto y) {  
            return binary(x, y);  
        };  
    };  
};
```

```
auto a = applyf(mul);  
auto b = a(3);  
auto c = b(4) // 12
```

Write Lambda function

- Write a function that takes a function and an argument and returns a function that takes the second argument and applies the function.

```
curry (mul , 3) (4)    // 12
```


Write Lambda function

```
auto curry = [] (auto binary, auto x) {  
    return [=] (auto y) {  
        return binary(x, y);  
    };  
};
```

```
curry(mul, 3) (4) // 12
```

Currying

- Currying is the technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions, each with a single argument.
- In lambda calculus functions take a single argument only.
- Must know Currying to understand Haskell.
- Currying != Partial function application.

Partial Function of Application

```
auto addFour = [] (auto a, auto b,  
                  auto c, auto d) {  
    return a+b+c+d;  
};  
auto partial = [] (auto func, auto a,  
                  auto b) {  
    return [=] (auto c, auto d) {  
        return func(a, b, c, d);  
    };  
};  
  
partial(addFour, 1, 2) (3, 4); //10
```

Write Lambda function

- Without creating a new function show 3 ways to create the inc function.

```
inc(4) // 5
```

Write Lambda function

```
auto inc = curry(add, 1) ;  
auto inc = addf(1) ;  
auto inc = applyf(add) (1) ;  
  
inc(4) // 5
```

Write Lambda function

- Write a function `composeu` that takes two unary functions and returns a unary function that calls them both.

```
composeu (inc, curry (mul, 5)) (3) // 20
```

Write Lambda function

```
auto composeu =[] (auto f1, auto f2) {  
    return [=] (auto x) {  
        return f2(f1(x));  
    };  
};
```

```
composeu(incl, curry(mul, 5))(3) // 20
```

Write Lambda function

- Write a function that returns a function that allows a binary function to be called exactly once.

```
once (add) (3, 4) // 7
```

```
once (add) (3, 4) // error
```


Write Lambda function

```
auto once = [] (auto binary) {  
    bool done = false;  
    return [=] (auto x, auto y) mutable {  
        if(!done) {  
            done = true;  
            return binary(x, y);  
        }  
        else  
            throw std::runtime_error("once!");  
    };  
};  
  
once(add) (3, 4) // 7  
once(add) (3, 4) // exception
```

Write Lambda function

- Write a function that takes a binary function and returns a function that takes two arguments and a callback and invokes the callback on the result of the binary function.

Write Lambda function

```
auto binaryc = [] (auto binary) {  
    return [=] (auto x, auto y, auto callback) {  
        return callback(binary(x,y));  
    };  
};
```

```
binaryc(mul) (5, 6, inc) // 31  
binaryc(mul) (5, 6, [] (int a) {  
    return a+1;  
});
```

Write 3 functions

- Unit
 - same as identityf
- Stringify
 - that stringifies its argument and applies unit to it.
- Bind
 - that takes a result of unit and returns a function that takes a callback and returns the result of callback applied to the result of unit

Write 3 functions

```
auto unit = [] (auto x) {  
    return [=] () { return x; };  
};
```

```
auto stringify = [] (auto x) {  
    std::stringstream ss;  
    ss << x;  
    return unit(ss.str());  
};
```

```
auto bind = [] (auto u) {  
    return [=] (auto callback) {  
        return callback(u());  
    };  
};
```

Verify

```
std::cout << "Left Identity "  
           << stringify(15) ()  
           << "=="  
           << bind(unit(15)) (stringify) ()  
           << std::endl;
```

```
std::cout << "Right Identity "  
           << stringify(5) ()  
           << "=="  
           << bind(stringify(5)) (unit) ()  
           << std::endl;
```

Applying Lambdas

- Design your APIs with lambdas in mind
 - Usually through template parameters
- Invest in re-learning STL algorithms

```
template <typename Callback>
void enum_windows(const string& title, Callback callback) {
    . . . callback(current_window);
}
```

```
template <typename RAIter, typename Comparer>
void superfast_sort(RAIter from, RAIter to, Comparer cmp);
```

Lambdas and Function Pointers

- Stateless lambdas are convertible to function pointers!
 - Useful for interop with C-style APIs that take function pointers (e.g., Win32)

```
ReadFileEx(file, buffer, 4096, &overlapped,  
    [](DWORD ec, DWORD count, LPOVERLAPPED overlapped) {  
        // process the results  
    })  
);
```


Today's learning objectives

- Functional programming
- Introduction to Lambda functions
- Playing with Lambda functions
- "New" features of C++11
- What's new in C++ 14?

Range-Based for Loop

- Automatic iterator over arrays and STL collections
 - Your collection will work – provide `begin()`, `end()`, and an input iterator over the elements
 - Can also specialize global `std::begin()`, `std::end()` if you don't control the collection class

```
int numbers[] = ...;
for (int n : numbers) std::cout << n;
std::map<std::string, std::list<int>> M;
for (const auto& pair : M)
    std::cout << pair.first << " ";
for (auto n : pair.second)
    std::cout << n << " ";
```

Initializer Lists

- Initialize arrays, lists, vectors, other containers— and *your own* containers—with a natural syntax
- Also applies to structs/classes!

```
vector<int> v { 1, 2, 3, 4 };  
list<string> l = { "Tel-Aviv", "London" };  
my_cont c { 42, 43, 44 };  
point origin { 0, 0 }; //not a container, but has ctor taking two ints
```

```
class my_cont {  
    public: my_cont(std::initializer_list<int> list) {  
        for (auto it = list.begin(); it != list.end(); ++it) . . .  
    }  
};
```

Delegating Constructors

- No more `init()`-functions! You can now call a constructor from another constructor

```
class file {  
public:  
    file(string filename) : file(filename.c_str()) {}  
    file(const char* filename) : file(open(filename)) {}  
    file(int fd) {  
        // Actual initialization goes here  
    }  
};
```

Alias Templates

- typedefs can now be templates
- The using keyword can replace typedef completely

```
template <typename K, typename V, typename Alloc>  
class map;
```

```
template <typename T>  
using simple_set<T> = map<T, bool, DefaultAllocator>;
```

```
using thread_func = DWORD(*)(LPVOID);
```

Non-Static Data Member Initializers

- Data members can be initialized inline
- The compiler adds the appropriate code prior to each constructor

```
class counter {  
    int count = 0;  
    SRWLOCK lock { SRWLOCK_INIT };  
public:  
    // ...  
};
```

More Miscellaneous Features

- Explicit conversion operators
- Raw string literals
- Custom literals

```
auto utf8string  = u8"Hello";  
auto utf16string = u"Hello";  
auto utf32string = U"Hello";
```

```
auto raw = R"(I can put " here and also \)";  
auto break_time = 5min; // in the STL as of C++14
```

New Smart Pointers

- STL now has three types of smart pointers, eliminating the need to ever use delete
 - If you are the sole owner of the object, use `unique_ptr` to make sure it's deleted when the pointer dies (RAII)
 - If you want to share the object with others, use `shared_ptr`—it will perform smart reference counting
 - If you got yourself a cycle, use `weak_ptr` to break it!

unique_ptr

- Sole owner of an object
 - Moveable, not copyable
- Replaces auto_ptr (which can be copied, but leads to problems)

```
unique_ptr<expensive_thing> create() {  
    unique_ptr<expensive_thing> p(new expensive_thing);  
    // ...do some initialization, exceptions are covered by RAII  
    return p;  
}
```

```
unique_ptr<expensive_thing> p = create();    // move constructor used!
```

```
//another example is storing pointers in containers:
```

```
vector<unique_ptr<string>> v = { new string('A'), new string('B') };
```

shared_ptr

- Thread-safe reference-counted pointer to an object with shared ownership
 - When the last pointer dies, the object is deleted

```
struct file_handle {  
    HANDLE handle;  
    file_handle(const string& filename) ...  
    ~file_handle() ... //closes the handle  
};  
class file {  
    shared_ptr<file_handle> _handle;  
public:  
    file(const string& filename) : _handle(new file_handle(filename)) {}  
    file(shared_ptr<file_handle> fh) : _handle(fh) {}  
}; //can have multiple file objects over the same file_handle
```

weak_ptr

- Points to a shared object but does not keep it alive (does not affect reference count)
- Breaks cycles between shared_ptrs

```
class employee {
    weak_ptr<employee> _manager;
    vector<shared_ptr<employee>> _direct_reports;
public:
    void beg_for_vacation(int days) {
        if (auto mgr = _manager.lock()) { mgr->beg(days); }
        else { /* your manager has been eliminated :-) */ }
    }
};
```

Guidance for Smart Pointers

- Own memory with smart pointers: don't delete
- Allocate shared pointers with `make_shared`
- Most owners of pointers can use `unique_ptr`, which is more efficient than `shared_ptr`
- Don't use smart pointers for passing parameters if the usage lifetime is a subset of the function call's lifetime
- Pass `unique_ptr<T>` by value to a sink that retains the object pointer and will destroy it
- Pass `shared_ptr<T>` by value to express taking shared ownership of the object

Guidance for Smart Pointers:

Examples

```
struct window {  
    // Okay - takes ownership of the window, will destroy it.  
    void set_owned_child(unique_ptr<window> window) {  
        child = std::move(window);  
    }  
    unique_ptr<window> child;  
};
```

```
// Weird - use plain & or * instead.  
currency calculate_tax(const unique_ptr<employee>& emp);
```

```
// Okay - takes shared ownership of the callback,  
// lifetime has to be extended past the call site  
void set_callback(shared_ptr<callback> cb);
```

Today's learning objectives

- Functional programming
- Introduction to Lambda functions
- Playing with Lambda functions
- "New" features of C++11
- What's new in C++ 14?

In C++ 14...

In C++ 14, you write

```
auto auto(auto auto) { auto; }
```

and the compiler infers the rest from context.

Generic Lambdas

- Lambda parameters can be declared with the auto specifier
- This is not type deduction – this is a template!

```
auto lambda = [](auto f) { f.foo(); };
```

// equivalent to:

```
struct __unnamed__ {  
    template <typename S>  
    void operator()(S f) const { f.foo(); }  
};  
auto lambda = __unnamed__();
```


Lambda Capture Extensions

- Lambdas can now capture arbitrary expressions, which enables renaming and capture-by-move semantics

```
std::async([tid = std::this_thread::get_id]() {  
    cout << "called from thread " << tid << endl;  
});
```

```
unique_ptr<window> top_level = ...;  
auto f = [top = std::move(top_level)]() {  
    // access "top"  
};
```

Function Return Type Deduction

- Functions can be declared as returning `auto`, and the compiler deduces the return type (if consistent)

// Not okay, return type unknown at recursive call:

```
auto fibonacci(int n) {  
    if (n != 1 && n != 2) return fibonacci(n-1) + fibonacci(n-2);  
    return 1;  
}
```

```
auto fibonacci(int n) { // Okay, return type deduced to be int  
    if (n == 1 || n == 2) return 1;  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```