Moving to Modern C++:
Lambdas In Depth

Steve Dewhurst
Dan Saks

1

Modern C++

The C++ programming language is defined by a formal international standard specification. That standard was updated in 2011 and again in 2014. Modern C++ is the language as specified by these recent standards.

Compared to the earlier standards, Modern C++ introduces a significant number of new language and library features. This course focuses primarily on the language features of Modern C++ and programming techniques that use those features.

2

Saks & Associates
393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-937-324-3601
dan@dansaks.com
www.dansaks.com

3

4

Legal Stuff

- These notes are Copyright © 2017 by Stephen C. Dewhurst and Daniel Saks.
- If you have attended this course, then:
  - You may make copies of these notes for your personal use, as well as backup copies as needed to protect against loss.
  - You must preserve the copyright notices in each copy of the notes that you make.
  - You must treat all copies of the notes — electronic and printed — as a single book. That is,
    - You may lend a copy to another person, as long as only one person at a time (including you) uses any of your copies.
    - You may transfer ownership of your copies to another person, as long as you destroy all copies you do not transfer.

5

More Legal Stuff

- If you have not attended this course, you may possess these notes provided you acquired them directly from Saks & Associates, or:
  - You have acquired them, either directly or indirectly, from someone who has (1) attended the course, or (2) paid to attend it at a conference, or (3) licensed the material from Saks & Associates.
  - The person from whom you acquired the notes no longer possesses any copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

6

About Steve Dewhurst

Steve Dewhurst is the cofounder and president of Semantics Consulting, Inc. He is the author of the critically-acclaimed books *C++ Common Knowledge* and *C++ Gotchas*, and the co-author of *Programming in C++*. He has written numerous technical articles on C++ programming techniques and compiler design.

As a Member of Technical Staff at AT&T Bell Laboratories, Steve worked with C++ designer Bjarne Stroustrup on the first public release of the C++ language and cfront compiler. He was lead designer and implementer of AT&T's first non-cfront C++ compiler. As a compiler architect at Glockenspiel, Ltd., he designed and implemented a second C++ compiler. He has also written C, COBOL, and Pascal compilers.

Steve served on both the ANSI/ISO C++ standardization committee and the ANSI/IEEE Pascal standardization committee.

7

About Steve Dewhurst

Steve has consulted for projects in areas such as compiler design, embedded telecommunications, e-commerce, and derivative securities trading. He has been a frequent and highly-rated speaker at industry conferences such as *Software Development* and *Embedded Systems*. He was a Visiting Scientist at CERT and a Visiting Professor of Computer Science at Jackson State University.

Steve was a contributing editor for *The C/C++ User's Journal*, an editorial board member for *The C++ Report*, and a cofounder and editorial board member of *The C++ Journal*.

Steve received an A.B. in Mathematics and an Sc.B. in Computer Science from Brown University in 1980 and an M.S. in Engineering/ Computer Science from Princeton University in 1982.

8

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan is a contributing editor for *embedded.com* online. He has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught C and C++ to thousands of programmers around the world. He has presented at conferences such as *Software Development*, *Embedded Systems*, and *C++ World*. He has served on the advisory boards of the *Embedded Systems* and *Software Development* conferences.

9

About Dan Saks

Dan served as secretary of the ANSI and ISO C++ standards committees and as a member of the ANSI C standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. Previously, he was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/Information Science from Case Western Reserve University.

10

Past C++ Standards

- ***1998***: "C++98"
  - the first international C++ standard (ISO [1998])
- ***2003***: "C++03"
  - a revised international C++ standard (ISO [2003])
  - bug fixes
  - nothing else new
- ***2005***: "TR1"
  - Library "Technical Report 1" (ISO [2005])
  - proposals for library extensions
  - not a new standard

11

Modern C++ Standards

- ***2011***: "C++11"
  - a new international C++ standard (ISO [2011a])
  - significant new language features
  - most of TR1, plus more library components
- ***2014***: "C++14"
  - the latest international C++ standard (ISO [2014])
  - mostly improvements to C++11 features
  - a few new features, too

12

Lambdas

Advance Warning!

- We're about to examine lambdas in more detail than is probably healthy.
- In the following, let's keep in mind the intended purpose of lambdas as stated in the C++ standard (ISO [2014], 5.1.2):

✓ *"Lambda expressions provide a concise way to create simple function objects."*

- Lambdas don't <u>have</u> to be "simple," but they typically <u>should</u> be.

STL Algorithms and Predicates

- The standard algorithm `count(b, e, v)` returns the number of elements in `[b, e)` that are equal to `v`.
- For example, the following call to `count` returns the number of elements in scores equal to 100:

```
vector<int> scores;     // test scores
~~~
n = count(scores.begin(), scores.end(), 100);
```

- `count` uses == as the comparison operator.

15

STL Algorithms and Predicates

- `count_if` is a more flexible variant of `count`.
- `count_if(b, e, p)` returns the number of elements in `[b, e)` for which unary predicate `p` is true.
- A ***predicate*** is a function or function-like object that returns a boolean (or something convertible to boolean) indicating whether a certain condition is true for its operand(s).
- Different predicates accept different numbers of arguments:
  - A ***unary predicate*** accepts just one argument.
  - A ***binary predicate*** accepts two.
  - A ***ternary predicate*** accepts three.
- For example…

16

STL Algorithms and Predicates

- Schools in the United States commonly translate numeric test scores into letter grades:
  - "A" (excellent), "B", "C", "D", and "F" (failing).
- Suppose a score that's greater than or equal to 93 is an "A":
- Here's a predicate function you can use with `count_if` to count up the "A"s:

```
bool is_an_A(int s) {
    return s >= 93;
}
~~~
n = count_if(scores.begin(), scores.end(), is_an_A);
```

17

Functional Classes

- Alternatively, you can implement `is_an_A` as a ***functional class*** or ***function object type***:

```
struct is_an_A: unary_function<int, bool> {
    bool operator()(int s) const {
        return s >= 93;
    }
};
```

- Using this functional class, the call to `count_if` looks like:

```
n = count_if(scores.begin(), scores.end(), is_an_A());
```

- Note that the `unary_function` base class has been deprecated.

18

Function Objects

- Here, the expression `is_an_A()` creates a default-constructed temporary function object:

```
n = count_if(scores.begin(), scores.end(), is_an_A());
```

- It's a shorthand for declaring a default-constructed named function object, as in:

```
is_an_A temp;
n = count_if(scores.begin(), scores.end(), temp);
```

19

Functional Class Templates

- The standard header `<functional>` provides numerous class templates you can use to compose function objects on the fly.
- For example:
  - `greater_equal<T>()(x, y)` returns true if `x >= y`.
  - `bind2nd<int>(greater_equal<int>(), 93)(x)` returns true if `x >= 93`.
    - `bind2nd` is now deprecated.
- Thus, you can count the number of "A"s using:

```
n = count_if(
        scores.begin(), scores.end(),
        bind2nd<int>(greater_equal<int>(), 93)()
    );
```

20

Functional Class Composition

- Using functional class templates gets complicated in a hurry.
- For example, suppose a score that's greater than or equal to 84 and less than or equal to 92 is a "B".
- The following call to `count_if` counts the number of "B"s:

```
n = count_if(
        scores.begin, scores.end(),
        compose2(           // non-standard, but common
            logical_and<bool>(),
            bind2nd(greater_equal<int>(), 84),
            bind2nd(less_equal<int>(), 92)
        )()
    );
```

21

Lambda Expressions

- Modern C++ provides ***lambda expressions*** (or just ***lambdas***) as a simpler way to create function objects.
- For example, here's a lambda you can use to count the "A"s:

```
n = count_if(
        scores.begin(), scores.end(),
        [](int s) { return s >= 93; }       // lambda
    );
```

- The lambda above yields a temporary default-constructed function object…

22

Lambda Closures

- Evaluating a lambda yields a temporary function object called a ***closure object***.
- That object is an rvalue.
- It's type is called the ***closure type***.
- For this lambda expression:

```
[](int s) { return s >= 93; }
```

the closure type is a class more-or-less defined as:

```
class __closure_type {      // compiler-generated name
public:
    bool operator ()(int s) const { return s >= 93; }
};
```

23

- Using a lambda lets you simplify this:

```
n = count_if(
        scores.begin, scores.end(),
        compose2(
            logical_and<bool>(),
            bind2nd(greater_equal<int>(), 84),
            bind2nd(less_equal<int>(), 92)
        )()
    );
```

to this:

```
n = count_if(
        scores.begin(), scores.end(),
        [](int s) { return (s >= 84) && (s <= 92); }
    );
```

24

Anatomy of a Lambda

- The syntax of lambda expressions is straightforward, if odd:

```
[](int s) { return s >= 93; }
```

compound-statement

lambda-declarator (optional)

lambda-introducer

- The minimal lambda expression is therefore:

```
[]{}
```

- It does nothing and returns.

Lambda Return Type

- If the compound-statement contains a simple return, the return type is the type of the return expression.
- Alternatively, you can specify a trailing return type:

```
[](int s) -> int {      // -1, 0, or 1
    if (valid(s))
        return s >= 93;
    return -1;
}
```

- In C++14, the return type of the lambda can be deduced from a more complex body, provided all return expressions have the same type.

Closure Types and Copying

- Generally, use auto to declare a variable to hold a closure object:

```
auto clo1 = [](){};  // my favorite lambda...
```

- Each lambda expression has a unique, unnamed closure type:

```
auto clo2 = [](){};  // decltype(clo1) != decltype(clo2)
```

- Closures may be copy-initialized, but they have no default constructor:

```
auto clo3 = clo1;     // OK to copy construct
decltype(clo1) clo4; // error! no default ctor
```

27

Lambda Comparators

- The absence of default lambda initialization can be problematic when using lambdas as comparators, equality operators, or hash functions for associative containers:

```
auto comp = [](T const &a, T const &b)
    { return a.level() > b.level(); };
~~~
set<T, decltype(comp)> tset;    // error! no default init
```

- You must use copy initialization:

```
set<T, decltype(comp)> tset (comp);     // OK
```

- Copy assignment for lambdas is not (supposed to be!) supported.

28

Lambdas vs. Function Pointers

- This is similar to using function pointers in this context:

```
bool comp(T const &a, T const &b) {
    return a.level() > b.level();
};
~~~
set<T, decltype(comp)> tset; // oops! default is nullptr!
set<T, bool (*)(T const &, T const &)> tset2; // same
```

- You must use non-default initialization:

```
set<T, decltype(comp)> tset (comp); // OK
set<T, bool (*)(T const &, T const &)> tset2 (comp);
```

29

Lambdas as Unevaluated Operands

- As with other definitions, lambda expressions may not appear as unevaluated operands:

```
cout << sizeof([]{ return 12.3; });     // error!
decltype([]{ return 12.3; }) alambda;   // error!
```

- As with lambdas as set comparators, you use a workaround:

```
auto forsize = []{ return 12.3; };

cout << sizeof(forsize) << endl;        // probably 1
cout << sizeof(forsize()) << endl;      // probably 8
decltype(forsize) forsize2 = forsize;   // a lambda
decltype(forsize()) d = 12.34;          // a double
```

30

Access to Statics

- As explained earlier, a lambda can access explicitly-passed arguments.
- It can also access static names that are in scope:

```
extern int a = 10;
~~~
void aFunc() {
    static int b = 12;
    auto alambda = []{ return a + b; };
    ~~~
}
```

31

Capture

- Lambdas also have access to automatic variables in scope.
- However, access to these requires use of one or more ***captures*** in the lambda-introducer:

```
void selloff(vector<Stock> &s, double base) {
    auto to_sell = partition(begin(s), end(s),
        [base](Stock const &stock)
            { return stock.price() < base; });
    for_each(to_sell, end(s),
             [](Stock &s) { s.sell(); });
    s.erase(to_sell, s.end());
}
```

- This lambda *explicitly captures* the name base.

32

Capture Implementation

- Here, again, is the lambda expression with the capture:

```
[base](Stock const &stock)
      { return stock.price() < base; }
```

- Here's the equivalent hand-coded closure type:

```
class __closure_type {
public:
    bool operator ()(Stock const &stock) const
        { return stock.price() < base_; }
private:
    double base_;
};
```

33

Capture by Reference

- You can capture an automatic variable by reference, instead of by value:

```
vector<int> v { 1, 2, 3, 4 };
int total = 0;
for_each(begin(v), end(v),
    [&total](int a){ total += a; });
cout << "TOTAL: " << total << endl;     // displays 10
```

34

Capture by Reference

- Here's the lambda expression with a capture by reference:

```
[&total](int a){ total += a; }
```

- The equivalent hand-coded closure type defines `total` as a reference member:

```
class __closure_type {
public:
    void operator ()(int a) const
        { total_ += a; }
private:
    int &total_;
};
```

35

Reference Capture and Copying

- Reference capture causes the lambda's closure to contain a reference data member.
- Note that, according to the standard, all lambdas are supposed to have deleted copy assignment so a reference member is not a handicap.
- However, some compilers incorrectly permit lambda copy assignment... except if reference capture is used.

```
auto pred = [&base](Stock const &stock)
    { return stock.price() < base; };
auto to_sell = partition(begin(s), end(s), pred);
                                // copy init ^
auto pred2 = pred;              // copy init
pred2 = pred;                   // error! copy assign
```

36

Default Capture Modes

- A default capture mode is a shorthand notation for capturing an unbounded set of local automatic variables.
- For example, here's our earlier example:

```
[base](Stock const &stock)
    { return stock.price() < base; });
```

- It could be written as:

```
[=](Stock const &stock)
    { return stock.price() < base; });
```

- Using the = default capture mode will copy any referenced automatic local variable into the lambda's closure by value.

37

Default Capture Modes

- We can also use default reference capture.
- For example, here's an earlier example:

```
[&total](int a){ total += a; }
```

- It could be written as:

```
[&](int a){ total += a; }
```

- Using the & default capture mode will refer to any referenced automatic local variable into the lambda's closure by reference.

38

A Local Predicate

- Suppose we'd like to sell off underperforming stock:

```
void selloff(vector<Stock> &s, double base) {
    static double const multiplier = 6.02;
    size_t num_sold = 0;
    auto pred = ~~~;        // see next slide...
    auto to_sell = partition(begin(s), end(s), pred);
    for (auto i = to_sell; i != s.end(); ++i)
        i->sell();
    s.erase(to_sell, s.end());
}
```

39

Multiple Captures

- The lambda-introducer may contain a list of captures, and may combine default and individual capture modes.

```
void selloff(vector<Stock> &s, double base) {
    static double const multiplier = 6.02;
    size_t num_sold = 0;
    auto pred = [&, base](Stock const &stock) {
            if (stock.price() < base * multiplier) {
                ++num_sold;
                return true;
            }
            return false;
        };
    ~~~
}
```

40

Multiple Captures

```
[&, base](Stock const &stock) {
    if (stock.price() < base * multiplier) {
        ++num_sold;
        return true;
    }
    return false;
};
```

- This lambda refers to three local names:
  - `base` is explicitly referenced by value.
  - `multiplier` is static, and does not participate in capture.
  - `num_sold` is implicitly referenced by the default reference capture mode.

41

Implementation of Multiple Capture

```
class __closure_type {
public:
    bool operator ()(Stock const &stock) const {
        if (stock.price() < base_ * multiplier) {
            ++num_sold_;
            return true;
        }
        return false;
    }
private:
    double base_;
    size_t &num_sold_;
    // nothing for multiplier...
};
```

42

Explicit Capture and Ordering

- Officially, the order in which captured members appear in the lambda's closure class is unspecified.
- In practice, the order often reflects the order of the capture list.
- For example, the first lambda below may place the storage for `num_sold` before `base`.
- The second may do the reverse:

```
[&num_sold, base](Stock const &stock) { ~~~ }
[base, &num_sold](Stock const &stock) { ~~~ }
```

✓ *Avoid depending on the order of closure members.*

- However, you may use ordering to affect closure sizes in a platform-dependent manner.

43

Lambda Capture Examples

Introducer	Meaning
`[]`	No capture
`[a]`	Capture a by value
`[&a]`	Capture a by reference
`[=]`	Capture everything by value
`[&]`	Capture everything by reference
`[a, b, c]`	Capture a, b, and c by value
`[a, &b, c]`	Capture a and c by value, b by reference
`[=, &a]`	Capture everything by value, except a
`[&, a]`	Capture everything by reference, except a
`[a = init]`	Init-Capture (C++14)
`[this]`	Capture the members of this class object

44

Dangling Captures

- As always, we have to be concerned with lifetime issues.
- What if a closure object refers to data that's been destroyed or is inaccessible?

```
template <typename F>
void spawn_counted_daemon(size_t count, F op) {
    thread background ([&]{ while (count--) op(); });
    background.detach();
}
```

- In this case, it's likely that the lambda executed in the detached thread will still be active when spawn_counted_daemon returns.
- The most obvious problem is that the reference to count will dangle, though there are also likely to be issues with op.

45

Typical Closures are Const

- A quick fix might be to use a different default capture mode:

```
template <typename F>
void spawn_counted_daemon(size_t count, F op) {
    thread background ([=]{ while (count--) op(); });
    background.detach();
}
```

- But…

46

Typical Closures are Const

- ...this won't compile:

```
class __closure_type {
public:
    void operator ()() const { while (count_--) op_(); }
private:
    size_t count_;
    F op_;
};
```

- By default, the operator() in a closure class is a const member function.

47

Mutable Lambdas

- A ***mutable lambda*** has a non-const operator():

```
[=]() mutable { while(count--) op(); }
```

- This gives a more appropriate implementation in this case.

```
class __closure_type {
public:
    void operator ()() { while (count_--) op_(); }
private:
    size_t count_;
    F op_;
};
```

- Note: C++17 permits constexpr lambdas.

48

Alternatives to Mutable Lambdas

- An alternative is to use a modifiable local variable:

```
[=]() { size_t local = count; while (local--) op(); }
```

- It produce a closure class such as:

```
class __closure_type {
public:
    void operator ()() const {
        size_t local = count; while (local--) op_();
    }
private:
    size_t count_;
    F op_;
};
```

49

Lambdas in Member Functions

- *Ecce* employee:

```
class Employee {
public:
    ~~~
    double pay_amount() const;
    double pay_rate() const;
private:
    ~~~
    vector<double> hours_worked_;
    double annoyance_factor_;
};
```

50

Lambdas in Member Functions

- Employees must be paid:

```
double Employee::pay_amount() const {
    auto calc
        = [this](double a, double hours) {
            return a + (hours * pay_rate())
                / annoyance_factor_;
        };
    return accumulate(
        begin(hours_worked_), end(hours_worked_),
        0.0, calc
    );
}
```

51

Lambdas in Member Functions: Access

- This lambda function occurs in the scope of a member function:

```
auto calc
    = [this](double a, double hours) {
        return a + (hours * pay_rate())
            / annoyance_factor_;
    };
```

- The lambda is part of a member function implementation.
- It therefore has access to annoyance_factor_, a private member of its enclosing class.

52

Implementation

- The lambda closure is implemented like this:

```
class __closure_type {
public:
    double operator ()(double a, double hours) const {
        return a + (hours * this_->pay_rate())
            / this_->annoyance_factor_;
    }
private:
    Employee *this_;
};
```

- Members pay_rate and annoyance_factor_ aren't captured.
- Rather, a copy of the this pointer is captured.

53

The Meaning of this in a Lambda

- Any use of this in the body of a lambda refers to an enclosing (non-lambda!) object, not to the lambda's closure.
- This is one of several important differences between lambdas and similar hand-generated function objects.

```
auto lamb = [] { cout << this << endl; } // error!

struct Lamb {
    void operator ()() const
        { cout << this << endl; }        // OK
};
```

- Access to a closure's this pointer would be of limited utility, since we don't know the names of the closure's members.

54

Member Capture

- Consider a different approach to member capture.

```
[](double a, double hours) { ~~~ }  // error!
```

- In this case, no capture is specified, and the member names are not found.

```
[=](double a, double hours) { ~~~ } // OK
```

- In this case, we're using default copy capture, and the lambda will capture a copy of the `this` pointer.
- It won't capture the members directly, because they're not local automatic variables.

55

Member Capture

- If you're concerned about a possible dangling `this` pointer, you should capture a local copy of the members of interest:

```
double annoyance_factor = annoyance_factor_;
double pay_rate = this->pay_rate();
auto calc = [annoyance_factor, pay_rate]
    (double a, double hours) {
        return a + (hours * pay_rate) / annoyance_factor;
    };
```

56

Reaching Scope

- Note another important difference between a lambda and a similar hand-coded function object.
- The lambda may refer to names in enclosing scopes up to the innermost enclosing function body. This is the lambda's "reaching scope."

```
double Employee::pay_amount() const {
    struct Calc {
        auto operator ()(double a, double hours) {
            return a + (hours * pay_rate()) // error!
                / annoyance_factor_;        // error!
    };
    return accumulate(
        begin(hours_worked_), end(hours_worked_),
        0.0, Calc());
}
```

57

Capture Morality

- Meyers [2015] recommends avoiding default capture modes, largely because they can lead to dangling references and dangling pointers.
- Explicit capture forces you to consider the viability of each capture.

```
[=](double a, double hours) { ~~~ }    // captures this
                                       // and what else?
[this](double a, double hours) { ~~~ } // captures this

[&]{ while (count--) op(); }           // deadly
[&count]{ while (count--) op(); }      // clearly deadly
```

58

Variadic Captures

- Note that it's possible to perform an explicit capture of a variadic argument pack.

```
template <typename... Ts>
void variad(Ts &&... args) {
    auto local_lambda = [args...]() {
        smooth(args...);
        soothe(move(args)...);
    };
    ~~~
}
```

59

Variadic Captures

- Default capture of the pack is possible, but not preferred.

```
template <typename... Ts>
void variad(Ts &&... args) {
    auto local_lambda = [=](){       // or [&]
        smooth(args...);
        soothe(move(args)...);
    };
    ~~~
}
```

60

Lambdas and Function Pointers

- Very simple lambdas define a conversion function to an appropriately-typed pointer to function.
- A lambda with no capture and referring only to its arguments and statics has such a conversion.
- For example:

```
auto ticker = [](string const &msg) {
    while (true) {
        this_thread::sleep_for(chrono::seconds(1));
        cout << msg << endl;
    }
}
```

61

Simple Lambda Conversion Operator

- The lambda's closure class looks something like:

```
class __closure_type {
public:
    void operator ()(string const &msg) const {
        while (true) {
            this_thread::sleep_for(chrono::seconds(1));
            cout << msg << endl;
        }
    }
    using __conv = void (*)(string const &);
    operator __conv() const;
};
```

62

Lambda Conversions

- For example:

```
auto ticker2 = ticker;                          // copy
void (*ticker3)(string const &) = ticker;   // conversion
using FP = void(*)(string const &);
FP ticker4 = ticker.operator FP();          // conversion
~~~
thread t (ticker4, "Tick!");            // call function
t.detach();
ticker2("Tock!");                       // call lambda
```

63

std::function

- `std::function` is a function object that may hold another "C++ callable entity" like a function pointer or function object.
- Lambdas generate function objects, and can be held by a `function`.
- The specialization of a `function` specifies the parameter and return types:

```
function<void (int)> func;
```

- This `function` object can hold a callable entity that accepts an argument that can be initialized by an `int` and returns something that's convertible to `void`.

64

std::function's Flexibility

- For example:

```
void f(int a) { ~~~ }
~~~
func = f;
func(123);  // calls f
```

- But also:

```
void *g(char c) { ~~~ }
~~~
func = g;
func(123);  // calls g
```

65

Flexibility

- A function function object can wrap a callable entity that
  - has a return type that can initialize the function's declared return type, and
  - has argument types that can be initialized by the function's declared argument types.

```
          function<void (int)> func;

void f(int a);          void *g(char c);
```

- A function provides a uniform interface for differently-typed callable entities.

66

Type Erasure

- function's implementation uses a technique called "type erasure."
- Type erasure allows a function object to hold callable entities of different types.
- function is a common choice to hold lambdas, since each lambda has a different type:

```
func = [](int a) -> void { ~~~ };       // 1) OK
func = [](double d) -> char { ~~~ };    // 2) also OK
func = [](string s) { ~~~ }             // 3) not OK!
```

- (3) fails because there's no conversion from int to string.
- Type erasure is flexible, but can be expensive.
- It often involves memory allocation and virtual functions.

67

Containers of Callbacks

- For example, we could have a collection of callbacks of different types through use of a container of functions:

```
class ResponsiveObject {
public:
    using CB = function<void(ResponsiveObject &)>;
    ResponsiveObject(string const &label);
    string const &label() const;
    void attach(CB const &callback);
    void attach(CB &&callback);
    void fire();
private:
    string label_;
    vector<CB> callbacks_;
};
```

68


```
class ResponsiveObject {
public:
    using CB = function<void(ResponsiveObject &)>;
    ResponsiveObject(string const &label)
        : label_(label) {}
    string const &label() const
        { return label_; }
    void attach(CB const &callback)
        { callbacks_.push_back(callback); }
    void attach(CB &&callback)
        { callbacks_.push_back(move(callback)); }
    void fire()
        { for (auto &cb : callbacks_) cb(*this); }
private:
    string label_;
    vector<CB> callbacks_;
};
```

69

Using Type Erasure

- The container of functions can accept any conformant callable object, like lambdas of different types.

```
ResponsiveObject ro ("Press");
ro.attach([](ResponsiveObject &o){ ~~~ });
auto other
    = [local1, local2](ResponsiveObject &o) { ~~~ };
ro.attach(other);
ro.attach([](ResponsiveObject &) { cout << "end\n"; });
~~~
if (pressed)
    ro.fire();
```

70

Recursive Lambdas

- Recursive lambdas are problematic.

```
auto fib = [&fib](int n) {
    return (n <= 2) ? 1 : fib(n-1) + fib(n-2);  // error
};
```

- This won't compile because the compiler can't deduce `fib`'s type until it ascertains the lambda's type.
- By that time, it's too late to write a directly-recursive implementation of `fib`.

71

Recursive Lambdas and `std::function`

- A depressingly-common suggestion is to introduce a level of indirection through `std::function`'s type erasure to permit the recursion.

```
std::function<int (int)> fib = [&fib](int n) {
    return (n <= 2) ? 1 : fib(n-1) + fib(n-2);
};
```

- This works.

✓ *Don't do it.*

- Introducing an intermediate `std::function` and its attendant type erasure mechanism introduces significant runtime cost.

72

Alternatives to Recursive Lambdas

- A lambda is just a convenient way to write a function object.
- A more efficient approach may be to just write a function object with a known type:

```
struct Fib {
    int operator ()(int n) const {
        return (n <= 2) ? 1 : Fib()(n-1) + Fib()(n-2);
    }
};
```

- This runs about 10 times faster than the previous version.

✓ *Lambdas generate function objects. If a lambda gives you problems, write the function object by hand.*

73

Alternatives to Recursive Lambdas

- In this simple case, we could use an inline function instead of an inline function object:

```
inline int fib(int n) {      // same efficiency as Fib
    return (n <= 2) ? 1 : fib(n - 1) + fib(n - 2);
}
```

- Declare it constexpr if there's any possibility that the argument will be a compile-time constant:

```
constexpr int fib(int n) {  // may be compile-time
    return (n <= 2) ? 1 : fib(n - 1) + fib(n - 2);
}
```

74

Algorithms vs. Loops

- Many standard algorithms are essentially fancy ways of looping through sequences:

```
for_each(begin(cont), end(cont),
         [](auto &a) { munge(preprocess(a)); });
```

- For simple traversals, we can make the loop explicit and process the sequence elements within the loop body:

```
for (auto &a : cont) {
    munge(preprocess(a));
}
```

75

Algorithm + Lambda vs. For + Body

- Stroustrup [2013] observes that often a task can often be implemented as a "choice between 'algorithm plus lambda' and 'for-statement with body.'"
- He suggests the choice be based on extensibility and maintainability considerations.
- Performance in either case is typically equivalent.
  - For long random access sequences the algorithm may give better performance through use of loop unwinding.
  - In C++17, some standard algorithms have parallel versions that may give better performance than a hand-coded loop.

76

Lambda Wrappers for Inlining

- An inline function won't be inlined if it's passed as an argument to an algorithm:

```
inline bool comp(T const &a, T const &b)
    { return a.val() > b.val(); }
~~~
sort(begin, end, comp);             // not inlined
```

- Using ptr_fun will wrap a pointer to the function, not the function:

```
sort(begin, end, ptr_fun(comp));    // not inlined
```

77

Keeping it Inline

- Wrapping the function in a uniquely-typed wrapper will allow the compiler to inline it:

```
sort(begin, end,        // inlined
     [](T const &a, T const &b) { return comp(a, b); });
```

- In C++03, you can write the wrapper by hand:

```
struct Comp: binary_function<T, T, bool> {
    bool operator ()(T const &a, T const &b) const
        { return comp(a, b); }
};
sort(begin, end, Comp());
```

78

Lambda Wrappers for Pseudofunctions

- Preprocessor “pseudofunctions” are dangerous.
- It’s easy to go wrong with them:

```
#define twice(E) ((E)+(E))
~~~
int a = 10;
cout << twice(++a);     // undefined behavior
```

- Avoid them if possible.

79

Fixing Pseudofunctions

- You can’t use a pseudofunction to customize a generic algorithm:

```
int a[] = { 0xDEAD, 0xBEEF, 0xCAFE, 0xBABE };
transform(a, a + 4, a, twice);          // error
```

- You have to wrap the pseudofunction.
- Function objects make good wrappers:

```
int a[] = { 0xBAAD, 0xF00D, 0xDEAD, 0xD00D };
transform(a, a + 4, a, [](int v) { return twice(v); });
```

- This wrapping preserves efficiency while avoiding the hazards of the pseudofunction.

80

Wrapping Possible Pseudofunctions

- A particularly noxious case occurs when a component is implemented as an inline function on one platform, and as a pseudofunction on another:

```
string s ("New job: fix Mr. Gluck's hazy TV, PDQ!");
transform(s.begin(), s.end(), s.begin(), toupper);
```

- This compiles on some platforms, but not on others.
  - If `toupper` is a macro, it won't compile.
- If it does compile, the call to `toupper` will not be inlined.
  - `toupper` will be passed as a pointer to a function.

81

Fixing Possible Pseudofunctions

- The solution is, of course, to wrap:

```
string s ("The five boxing wizards jump quickly.");
transform(s.begin(), s.end(), s.begin(),
          [](char c) { return toupper(c); });
```

- This approach is portable.
- It will inline `toupper` whether it's an inline function or a pseudofunction.

82

Deleting

- Cleaning up containers of pointers is usually tedious:

```
template <typename Cont>
void clean(Cont &c) {
    using type = typename Cont::value_type;
    static_assert(is_pointer<type>::value,
        "must be container of pointers");
    for (auto const el: c)
        delete el;
}
~~~
vector<Widget *> vw;
~~~
clean(vw);            // amateurish, should use RAII...
```

83

Template Members vs. Member Templates

- In C++03, we can use a generic algorithm and a deleter:

```
template <typename T>
struct Delete {
    void operator ()(T const *p) const
        { delete p; }
};

for_each(vw.begin(), vw.end(), Delete<Widget>());
```

- You must explicitly specialize the `Delete` class template to use it.
- There's no template argument deduction for class templates.
- Note: C++17 does permit some deduction for class templates.

84

Template Members vs. Member Templates

- Alternatively, the deleter may use a member template:

```
struct Delete {
    template <typename T>
    void operator ()(T const *p) const
        { delete p; }
};

for_each(vw.begin(), vw.end(), Delete());
```

- `Delete` is no longer a template; it doesn't require specialization.
- Instead, the compiler uses template argument deduction with `Delete`'s member template `operator ()`.

85

Universal Deletion

- Our universal deleter can delete anything that's deletable:

```
Delete d;
~~~
int *ip = new int(123);
d(ip);
~~~
Widget *wp = new Widget(sturm, drang);
d(wp);
```

86

Generic Lambdas in C++14

- A lambda that uses `auto` in its argument list is a ***generic lambda***:

```
auto stdop = [](auto &x) { x.fire(); x.forget(); }
```

- This lambda can handle any argument that can be fired and forgotten:

```
MyThread task(func);
stdop(task);
~~~
Employee emp;
stdop(emp);
~~~
Missile m;
stdop(m);
```

87

Non-Generic Lambda Implementation

- Recall that a non-generic lambda is implemented as a function object.
- Whereas a non-generic lambda like…

```
[](Missile &x) { x.fire(); x.forget(); }
```

…is probably implemented like this:

```
class __closure_type {
public:
    void operator ()(Missile &x) const
        { x.fire(); x.forget(); }
};
```

88

Generic Lambda Implementation

- A generic lambda like…

```
[](auto &x) { x.fire(); x.forget(); }
```

…is probably implemented like this:

```
class __closure_type {
public:
    template <typename T>
    void operator ()(T &x) const
        { x.fire(); x.forget(); }
};
```

- The `operator ()` is now a member template.

89

Deleting Again

- Our universal deleter…

```
struct Delete {
    template <typename T>
    void operator ()(T const *p) const
        { delete p; }
};
~~~
for_each(vw.begin(), vw.end(), Delete());
```

…is similar to this generic lambda:

```
for_each(vw.begin(), vw.end(), [](auto p) { delete p; });
```

90

"Handles" for Unspecified Types

- An ordinary generic function has an unspecified type, but we have a "handle" that refers to the type once it is specialized.

```
template <typename T>
void wrapper(T const &arg) {
    // arg is of type T const &, whatever T is...
    T temp = arg;
    ~~~
}
```

91

"Handles" for Unspecified Types

- A lambda in a template implementation has access to the type name, and so is not typically required to be a generic lambda.

```
template <typename T>
void wrapper(T const &arg) {
    // arg is of type T const &, whatever T is...
    T temp = arg;
    auto f = [](T &rtemp) { ... };
    ~~~
}
```

92

Problems With Generic Lambda Args

- A generic lambda also has an unspecified type, but we don't have a convenient "handle" that refers to the type once it is specialized.

```
auto wrapper = [] (auto const &arg) {
    // we don't have a name for arg's type
    ??? temp = arg;
    ~~~
};
```

93

An Often Too-Simple Solution

- We can use `decltype` to try to recover the type of the argument:

```
auto wrapper = [](auto const &arg) {
    decltype(arg) temp;  // ref to const...
    ~~~
    temp = arg;
};
```

- However, this simple approach may give surprising results for some argument types.

```
wrapper(1234); // errors!
```

- We'd like the deduced type for the `auto` placeholder, not the argument type.

94

An Auto Approach

- In this limited case we can use auto to recover the type of the template argument:

```
auto wrapper = [](auto const &arg) {
    auto temp = arg;  // auto removes ref and const...
    ~~~
};
```

- But it won't help in other situations:

```
auto wrapper = [](auto const *arg) {
    auto temp = arg;  // ptr and const still there...
    ~~~
};
```

95

Type Recovery

- Remember that auto is a placeholder for a type.
- In an argument declaration like auto const & we can determine an appropriate T by stripping off the reference and const.

```
auto wrapper = [](auto const *arg) {
    remove_cv_t<remove_pointer_t<decltype(arg)>> temp;
    ~~~
    temp = *arg;
};
```

- This will provide generally more appropriate behavior.

```
wrapper(1234); // OK...
```

96

Recovery Through Decay

- Often the `decay` facility of `<type_traits>` is appropriate.
- Decay performs the same type transformations that would occur in pass-by-value.

```
auto wrapper = [](auto const &arg) {
    using Temp = decay_t<decltype(arg)>;
    Temp temp;
    ~~~
    temp = arg;
};
```

97

Recovery for Forwarding

- The use of reference collapsing in the implementation of `std::forward` allows us to use `decltype` directly:

```
auto wrapper = [](auto &&arg) {
    f(forward<decltype(arg)>(arg));
};
```

98

Capture

- Let's revisit an earlier lambda:

```
[base](Stock const &stock) { ~~~ }
```

- It captures the local variable base by value and stores a copy in the closure object:

```
class __closure_type {
public:
    bool operator ()(Stock const &stock) const { ~~~ }
private:
    double base_;   // we don't know the member name
};
```

99

Init Capture in C++14

- C++14 lambdas permit ***init-capture***, such as:

```
[amt = base](Stock const &stock) { ~~~ }
```

- As before, it captures the local variable base by value and stores a copy in the closure object.
- But now we have a way to refer to the member in the closure:

```
class __closure_type {
public:
    bool operator ()(Stock const &stock) const { ~~~ }
private:
    double amt_; // now we can refer to the member as amt
};              // (but we still don't know the name)
```

100

Init Capture by Reference

- As with simple capture, init-capture can capture by reference:

```
[&amt = base](Stock const &stock) { ~~~ }
```

- This lambda captures the local variable base by reference.
- Again, we have a way to refer to the member in the closure type:

```
class __closure_type {
public:
    bool operator ()(Stock const &stock) const { ~~~ }
private:
    double &amt_;
};
```

101

Generalized Lambda Capture

- The initializer in an init-capture can be more than just a local variable.
- It can be an expression:

```
[var = expr]{ ~~~ }
```

- In the case, it declares a member in the closure type to which we can refer as var.
- It initializes the member as if we had written:

```
auto var = expr;
```

- That's a copy initialization. We can also use direct initialization syntax.

102

Generalized Capture and Scope

- A name declared in an init-capture is in the closure type's scope.
- However, the initializing expression is evaluated in the lambda's "reaching scope" — the enclosing scopes out to the innermost enclosing function:

```
[base = base * 1.02](Stock const &stock) { ~~~ }
```

- The first instance of identifier base refers to a closure member.
- The second instance refers to a name in the lambda's reaching scope.
- This is a departure from most of the rest of the C++ language, where a name comes into scope before its initializer is evaluated:

```
int a = a; // initialize a with itself...oops
```

103

Init Capture and Moving

- Possibly the most important aspect of init-capture is that it allows move operations into a closure.
- For example, without init-capture:

```
map<string, string> farm;   // a large farm
~~~
auto f1 = [farm] { ~~~ };   // by value; not cheap
auto f2 = [&farm] { ~~~ };  // by reference; maybe unsafe
```

- Of course, init-capture doesn't help us in and of itself:

```
auto f3 = [thefarm = farm] { ~~~ };     // by value
auto f4 = [&thefarm = farm] { ~~~ };    // by reference
```

104

Init Capture and Moving

- However, init-capture lets you specify an expression rather than just a simple local name:

```
map<string, string> farm;   // a large farm
~~~
auto f1 = [myfarm = move(farm)] { ~~~ };
```

- This lets you specify a move into a closure rather than a less efficient copy or potentially dangerous pass by reference.

105

Moving unique_ptr

- Suppose you want to move a `unique_ptr` into a lambda's closure:

```
using Farm = map<string, string>;
auto farm = make_unique<Farm>();
~~~
auto f1 = [farm] { ~~~ };    // error!
auto f1 = [&farm] { ~~~ };  // bad idea!
```

- You can't copy a `unique_ptr`, but you can move it.
- An init-capture allows an expression initializer:

```
auto f1 = [myfarm = move(farm)] { ~~~ };     // OK
```

- It use `std::move` to invoke `unique_ptr`'s move constructor:

106

Capture Timing

- Note that capture occurs when a lambda is defined, not when it is called.
- These semantics are particularly important in the presence of move operations:

```
auto f1
    = [myfarm = move(farm)] { ~~~ };  // move occurs here
~~~ // farm is in the "moved-from" state now
f1();                                // no move occurs
```

107

Factory Functions and Init Capture

- Recall our earlier `Shape` factory function:

```
unique_ptr<Shape> makeShape();
```

- There's no way to call a function in a simple capture:

```
auto processor = [makeShape()] { ~~~ }  // error!
```

- But you can call a function in an init-capture expression:

```
auto processor = [aShape = makeShape()] { ~~~ } // OK
```

108

Capture of Members

- Earlier, we were concerned about the possibility of a dangling `this` pointer.
- We solved this by captured local copies of the members:

```
double annoyance_factor = annoyance_factor_;
double pay_rate = this->pay_rate();
auto calc = [annoyance_factor, pay_rate]
    (double a, double hours) {
        return a + (hours * pay_rate) / annoyance_factor;
    };
```

109

Init Capture of Members

- Init-capture gives us the same capability without the need to declare local copies:

```
auto calc =
    [annoyance_factor = this->annoyance_factor_,
     pay_rate = this->pay_rate()]
    (double a, double hours) {
        return a + (hours * pay_rate) / annoyance_factor;
    };
```

110

Lambda Return Type Deduction

- C++11 lambdas can deduce return types for simple lambda bodies:

```
[](int arg) { return arg * arg - 1; }   // deduced int
```

- But not for more complex ones:

```
[](int priority) -> int {               // explicit int
    if (this_thread::get_id() == worker)
        return priority;
    else
        return -priority;
}
```

111

Enhanced Return Type Deduction in C++14

- In C++11, you can often rewrite the lambda using ? and :

```
[](int priority) {                      // deduced int
    return (this_thread::get_id() == worker)
           ? priority : -priority;
}
```

- Or, you can use C++14:

```
[](int priority) {                      // deduced int
    if (this_thread::get_id() == worker)
        return priority;
    else
        return -priority;
}
```

112

Nested Lambdas

- Lambdas may be defined within lambdas.
- The "reaching scope" of nested lambdas includes that of the enclosing lambdas.

```
auto remove_if_and_forget = [](auto &c, auto pred) {
    auto forgotten = 0;
    auto action = [&](auto el)
        { ++forgotten; el.forget(); };
    auto r = stable_partition(begin(c), end(c), pred);
    for_each(r, end(c), action);
    c.erase(r, end(c));
    cout << "Forgotten: " << forgotten << endl;
};
```

113

Variadic Lambdas

- A generic lambda may have an argument pack.

```
auto vgl = [](auto... args) noexcept
    { return sizeof...(args); };
```

- Like non-lambda variadic functions, a variadic lambda may be used as a "perfect forwarder."

```
auto forwardingLambda_var = [](auto &&... args) {
    preprocess(args...);
    finishoff(forward<decltype(args)>(args)...);
};
```

114

Lambdas vs. Function Objects

- We've noted that lambdas are similar to hand-written function objects.
- However, we have also noticed a number of differences:
  1. Use of `this` in a lambda body refers to an enclosing class object, not to the lambda's closure.
  2. We do not know the declared names of captured items.
  3. Lambdas have access to names in enclosing functions and enclosing lambdas through capture.
  4. Lambdas may capture private and protected members of an enclosing class.
  5. Lambdas have no copy assignment.
  6. Simple lambdas without capture may be implicitly converted to pointer-to-function.

115

116

Bibliography

- Abrahams [2010]. David Abrahams, Rani Sharoni, Doug Gregor, N3050=10-0040
- ISO [1998]. *ISO/IEC Standard 14882:1998, Programming languages—C++*.
- ISO [2003]. *ISO/IEC 14882:2003: Programming languages — C++*.
- ISO [2005]. *ISO/IEC TR 19768, C++ Library Extensions*.
- ISO [2011a]. *ISO/IEC 14882:2011: Programming languages — C++*.
- ISO [2011b]. *ISO/IEC 9899:2011: Programming languages — C*.
- ISO [2014]. *ISO/IEC Standard 14882:2014, Programming languages—C++*.
- Karlsson [2004]. Bjorn Karlsson, "The Safe Bool Idiom". *The C++ Source*. www.artima.com/cppsource/safebool.html

117

Bibliography

- Meyers [2015]. Scott Meyers, Effective Modern C++. O'Reilly.
- Stroustrup [2013]. Bjarne Stroustrup, *The C++ Programming Language, 4th ed*. Addison-Wesley.
- Sutter [2013]. Herb Sutter, "GotW #94 Solution: AAA Style (Almost Always Auto)", *Sutter's Mill*. herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/
- Sutter [2013a]. Herb Sutter, "GotW #91: herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/
- Sutter [2014]. Herb Sutter, "Back to the Basics! Essentials of Modern C++ Style", *CppCon*. www.youtube.com/watch?v=xnqTKD8uD64
- Vandevoorde [2003]. David Vandevoorde and Nicolai Josuttis, *C++ Templates*. Addison-Wesley.

118

119

120