Back to Basics
Lambda Expressions

Barbara Geller & Ansel Sermersheim
CppCon September 2020

Introduction

- Prologue
- History
- Function Pointer
- Function Object
- Definition of a Lambda Expression
- Capture Clause
- Generalized Capture
- This
- Full Syntax as of C++20
- What is the Big Deal
- Generic Lambda

Prologue

- Credentials
  - every library and application is open source
  - development using cutting edge C++ technology
  - source code hosted on github
  - prebuilt binaries are available on our download site
  - all documentation is generated by DoxyPress

  - youtube channel with over 50 videos
  - frequent speakers at multiple conferences
    - CppCon, CppNow, emBO++, MeetingC++, code::dive
  - numerous presentations for C++ user groups
    - United States, Germany, Netherlands, England

Prologue

- Maintainers and Co-Founders
  - CopperSpice
    - cross platform C++ libraries
  - DoxyPress
    - documentation generator for C++ and other languages
  - CsString
    - support for UTF-8 and UTF-16, extensible to other encodings
  - CsSignal
    - thread aware signal / slot library
  - CsLibGuarded
    - library for managing access to data shared between threads

Lambda Expressions

- History
  - lambda calculus is a branch of mathematics
    - introduced in the 1930's to prove if "something" can be solved
    - used to construct a model where all functions are anonymous

    - some of the first items lambda calculus was used to address
      - if a sequence of steps can be defined which solves a problem, then can a program be written which implements the steps
        - yes, always
      - can any computer hardware simulate any other computer
        - yes, given sufficient time and memory

    - languages which were influenced by lambda calculus
      - Haskell, LISP, and ML

Lambda Expressions

- History
  - why do we use the terminology lambda expression
    - greek letter λ refers to an anonymous function
    - lambda - chosen since it is equated with something nameless
    - expression - required since the code can be evaluated and will return a value
  - fundamental definition in C++
    - an expression which returns a function object
  - lambda expressions are in many computer languages
    - C++, C#, Groovy, Java, Python, Ruby

Lambda Expressions

- Function Pointer
  - data type
  - pointer to any function
  - signature of the function must match the declaration of the pointer
  - invoked by the pointer name just like a normal function call
  - a function pointer is not dereferenced

  - usage:
    - callback function
    - an argument to another function

Lambda Expressions

- Example 1
  - myProcess is a function pointer, where this pointer can only point to a function with a parameter of int and return type of void
  - std::exit is a function
    - name of a function implicitly converts to a function pointer

```
#include <cstdlib>

void (*myProcess)(int);     // declaration of the pointer
myProcess = std::exit;

myProcess(42);              // calls std::exit
```

Lambda Expressions

- Operator Overloading
  - any method which starts with “operator” followed by a symbol, are called overloaded operators
    - bool operator==(const T &value)
    - bool operator>(const T &value)
    - T operator+=(const T &value)
  - any method with the exact name “operator()” is called the function call operator
    - void operator()()
    - bool operator()(int value)
    - double operator()(double d1, double d2)

Lambda Expressions

- Function Object
  - function object data type
    - class or structure with a function call operator method
  - function object
    - an instance of a function object data type
    - a callable object
  - a function object is called using normal function syntax
    - can receive parameters
    - has a return type

Lambda Expressions

- Example 2
  - create a class named Ginger
    - contains a method named operator()
    - Ginger is a function object data type
  - usage A and usage B do the exact same thing

```
class Ginger {
  void operator()(std::string str);
};

Ginger widget;
widget.operator()(“hello”);          // line A
widget(“hello”);                     // line B
```

Lambda Expressions

- Avoid using the term Functor
  - in mathematical terms it is a function which
    - takes one or more functions as its arguments
    - returns a function as the result
  - functor is also defined in mathematical category theory
  - functional programming
    - defines it as a function which performs mapping operations
  - when C++ developers uses the word “functor” they usually are referring to a function object or a function object data type

Lambda Expressions

- Terminology Review
  - functor
    - please use “function object” if that is what you mean
  - function pointer
    - pointer which refers to a function rather than pointing to data
  - function object data type
    - class which declares the operator()() method
  - function object
    - instance of a function object data type
  - std::function
    - container, holds a single function pointer or a function object

Lambda Expressions

- Definition of a Lambda Expression
  - first introduced in C++11
  - syntax for a lambda expression consists of specific punctuation
    - `[] () {}`
  - key elements
    - [capture clause] (parameter list) `->` return type { body }
  - a lambda expression . . .
    - assignable to a variable whose data type is usually auto
    - defines a function object

Lambda Expressions

- Definition of a Lambda Expression
  - capture clause
    - variables which are visible in the body
    - capture can happen by value or reference
    - can be empty
  - parameter list
    - can be empty or omitted
  - return type
    - data type returned by the body, optional, normally deduced
  - body
    - contains the programming statements to execute
    - can be empty

Lambda Expressions

- Example 3
  - x is captured from the outer scope
  - nothing in the parameter list
  - quiz: what value is printed

```
int main()
{
  int x = 42;
  auto myLamb = [x] ( )
    {
      cout << "Hello from a lambda expression, value = " << x << endl;
    };

  x = 7;
  myLamb();
}
```

Lambda Expressions

- Example 4
  - x is captured from the outer scope
  - nothing in the parameter list
  - quiz: what value is printed

```
int main()
{
  int x = 42;
  auto myLamb = [&x] ( )
    {
      cout << "Hello from a lambda expression, value = " << x << endl;
    };

  x = 7;
  myLamb();
}
```

Lambda Expressions

- Picky Details
  - everything to the right of the equal sign is the lambda expression
  - result of this expression is assigned to our variable
    - expression is first evaluated
    - then myLamb is initialized
  - give some thought to your variable name

  - a closure is simply a function object . . .
    - which is returned from the evaluation of a lambda expression
    - myLamb contains the closure
    - deduced type is a “closure data type”

```
auto myLamb = [ ] ( ) { return 17; };
```

Lambda Expressions

- Capture Clause
  - by value
    - capture is by const value
    - only variables in the local scope or “this” can be captured
    - x will be copied into the function object
    - capture occurs when the lambda expression is evaluated
    - original variable does not need to stay alive
    - if any captured value will be modified in the body, the lambda expression must be declared mutable

```
auto myLamb = [x] ( ) mutable { return ++x; };
```

Lambda Expressions

- Capture Clause
  - by reference
    - an & is added to indicate capture by lvalue reference
    - it is not valid to capture by rvalue reference

    - capture occurs when the lambda expression is evaluated
    - ensure captured lvalue references remain alive for the entire lifetime of the closure

```
auto myLamb = [&x] ( ) { return ++x; };
```

Lambda Expressions

- Capture Clause
  - C++11
    - capture by **value** or **reference**
  - C++14
    - generalized capture was added

Lambda Expressions

- Capture Clause
  - generalized capture
    - capture is initialized by value
      - [varA = 10]
      - [varB = x]
    - capture is initialized by reference
      - [&varC = y]
      - y must be declared in the local scope
    - capture is initialized by move
      - [varD = std::move(z)]
      - move occurs when the lambda expression is evaluated

Lambda Expressions

- Capture Clause
  - C++11
    - [this]
    - captures this pointer by value
  - C++14
    - [self = *this]
    - capture *this object by value, initializes a new variable
  - C++17
    - [*this]
    - capture *this object by value

Lambda Expressions

- Capture Clause
  - default capture by **value**
  - captures all variables used in the body of the lambda expression
    - `auto myLamb = [=] ( ) { return x + m_data; };`
  - default capture by **reference**
  - captures all variables used in the body of the lambda expression
    - `auto myLamb = [&] ( ) { return x + m_data; };`
  - starting with C++20
    - default capture of this pointer *by value* has been deprecated

Lambda Expressions

- Capture Clause
  - C++ standard defines the result of evaluating a lambda expression which does not capture anything as a special kind of closure
    - special closure has no state so it can be implicitly converted to a function pointer
    - if you are calling a C function which wants a function pointer, you can pass a lambda with an empty capture clause

Lambda Expressions

- Parameter List
  - C++11
    - declarations for the arguments passed to the closure
    - default parameters were not permitted
  - C++14
    - parameters can have a data type of auto (generic lambda)
    - default parameters are supported

```
auto myLamb = [ ] (const std::string &data, uint max = 20)
     { return data.substr(0, max); };
```

Lambda Expressions

- Return Type Deduction
  - C++11
    - if you have more than one return statement you must specify the return type
  - C++14
    - if there is more than one return statement they must deduce to the exact same data type or it must be specified

```
auto myLamb = [] (bool sloppy) -> double {
    if (sloppy)  { return 3; }

    return 3.14;
  };
```

Lambda Expressions

- Full Syntax as of C++20
  - template parameters
    - added in C++20
    - same syntax used with a template function or method
  - these are equivalent
    - `(auto && . . .  args)`
    - `<typename . . .  Ts>(Ts && . . .  args)`

```
[capture clause]  <template parameters>  (parameter list)
  specifier  exception  attribute  -> return type  requires { body }
```

Lambda Expressions

- Full Syntax as of C++20
  - specifier
    - mutable (C++11)
    - constexpr (C++17)
      - constexpr can usually be deduced so this keyword is optional
    - consteval (C++20)

```
[capture clause]  <template parameters>  (parameter list)
  specifier  exception  attribute  -> return type  requires { body }
```

Lambda Expressions

- Full Syntax as of C++20
  - exception
    - noexcept
    - throw
      - deprecated in C++11

```
[capture clause]  <template parameters>  (parameter list)
  specifier  exception  attribute  -> return type  requires  { body }
```

Lambda Expressions

- Full Syntax as of C++20
  - attribute
    - functions can have attributes before the return type
      - nodiscard, deprecated, noreturn
    - not available for a lambda expression, pending proposal
    - function type attributes appear at the end of the declaration
      - gnu::cdecl, gnu::regcall
    - modifies the signature

```
[capture clause]  <template parameters>  (parameter list)
  specifier  exception  attribute  -> return type  requires  { body }
```

Lambda Expressions

- Full Syntax as of C++20
  - requires
    - adds a constraint on . . .
      - capture clause
      - template parameters
      - arguments passed in the parameter list
      - anything which can be checked at compile time
    - example: `requires std::copyable<T>`

```
[capture clause]  <template parameters>  (parameter list)
  specifier  exception  attribute  -> return type  requires  { body }
```

Lambda Expressions

- What is the Big Deal
  - lambda expressions . . .
    - code is typically easier to read
    - more convenient to write than a function object
    - can be invoked immediately, not saved to a variable
    - pass to another function or method using std::function
    - pass to a template using type deduction
    - works nicely with std::visit(), std::thread, and algorithms

Lambda Expressions

- What is the Big Deal
  - code you write
    - lambda expression defines a function object
  - compiler
    - your lambda expression is used to generate an internal function object data type
  - run time
    - constructor in the function object data type is called, produces a closure

Lambda Expressions

- Callback
  - Computer Science
    - block of executable code which is passed as an argument to some other code
  - C Language
    - function pointer
      - passed to another function as an argument
  - C++
    - function pointer, function object, or a closure
      - passed to another function or method as an argument

Lambda Expressions

- Using a Callback with STL Algorithms (1)
  - std::count_if
    - returns the number of integers in the vector whose value is > 5

```
std::vector<int> data{ 1, 15, 3, 9, 11 };

// example A - passing a free function as a function pointer
bool myCallback(int i)  {
  return i > 5;
}

int resultA = std::count_if(data.begin(), data.end(), &myCallback);

// example B - using a lambda expression
int resultB = std::count_if(data.begin(), data.end(), [](int i){ return i > 5;});
```

Lambda Expressions

- Using a Callback with STL Algorithms (2)
  - std::count_if
    - returns the number of strings in the vector which start with the character ch

```
int count_str_starting_with(const std::vector<std::string> &data, char ch)
{
  return std::count_if( data.begin(), data.end(),
    [ch](const std::string &str) { return ! str.empty() && str[0] == ch; } );
}
```

Lambda Expressions

- Example 5
  - how do you capture std::unique_ptr in a lambda expression?

    - use a generalized lambda capture to “move capture”
    - capturing a move only type means the closure is move only
      - myLamb can only be moved
      - move only types are not copyable

```
std::unique_ptr<Widget> myPtr = std::make_unique<Widget>();

auto myLamb = [ capturedPtr = std::move(myPtr) ] ( )
    { return capturedPtr->computeSize(); };
```

Lambda Expressions

- Example 6
  - declare a lambda expression
  - myLamb has an lvalue category since it has a name
  - received using a template or std::function

```
auto myLamb = [] (double data) { return int(data); };
doThingA(myLamb);
doThingB(myLamb);


template <typename T>                                  // example A
void doThingA(T arg1);

void doThingB(std::function<int (double)> arg2)  // example B
```

Lambda Expressions

- Example 7 (a)
  - std::map<Key, Value, Compare>
  - our struct will override the default Compare operation

```
struct MyCompare {
  bool operator()(const std::string &a, const std::string &b) const {
    return a.size() < b.size();
  }
};

std::map<std::string, int, MyCompare>
  myMapA = { {"orange", 45}, {"apple", 95},
             {"kiwi", 40},   {"grapefruit", 22} };
```

Lambda Expressions

- Example 7 (b)
  - our lambda expression will override the default Compare operation
  - passing the type for the Compare parameter is enough to default construct our std::map

```
auto myLamb = [] (const std::string &a, const std::string &b)
  { return a.size() < b.size(); };

std::map<std::string, int, decltype(myLamb)>
  myMapB = { {"orange", 45}, {"apple", 95},
             {"kiwi", 40},   {"grapefruit", 22} };
```

Lambda Expressions

- Generic Lambda
  - added in C++14
  - data type for at least one parameter must be auto
  - when the lambda expression is compiled
    - internal code for the function call operator will be a template

  - quiz: which auto is the . . .
    - “function template argument deduction”
    - “auto type deduction”

```
auto myLamb = [ ] (auto var1, int var2) { return var1 + var2; }
```

Lambda Expressions

- Structured Bindings
  - structured bindings make it easier to access elements of tuples, arrays, and other compound types

```
auto [x, y] = someFunction();                 // line A
auto myLamb = [x] () { return x + 7; };       // line B
```

  - capturing a structured binding was deemed invalid according to the standard, so line B does not compile as of C++17
  - workaround: use a generalized lambda capture [x = x]

  - resolved in C++20
  - gcc and MSVC both allow the capture
  - known issue, clang still reports an error and it should not

Lambda Expressions

- Summary
  - function object
    - class or struct which declares the operator() method
  - lambda expression
    - evaluated at run time and produces a function object
    - can be assigned to a named variable which stores the closure
  - key parts of a lambda expression
    - capture clause, parameter list, body
    - lifetime matters when capturing by reference
  - data type of the closure should be auto
  - generalized capture - capture by move
  - generic lambda - parameter list data type of auto or T

Presentations

- ❑ Why CopperSpice, Why DoxyPress
- ❑ Compile Time Counter
- ❑ Modern C++ Data Types (references)
- ❑ Modern C++ Data Types (value categories)
- ❑ Modern C++ Data Types (move semantics)
- ❑ CsString library (unicode)
- ❑ Multithreading in C++
- ❑ Multithreading using libGuarded
- ❑ Signals and Slots
- ❑ Templates in the Real World
- ❑ What's in a Container
- ❑ Modern C++ Threads
- ❑ C++ Undefined Behavior
- ❑ Regular Expressions
- ❑ Type Traits
- ❑ C++ Tapas (typedef, forward declarations)
- ❑ C++ Tapas (typename, virtual, pure virtual)
- ❑ Overload Resolution
- ❑ Futures & Promises
- ❑ Thread Safety
- ❑ Constexpr Static Const
- ❑ When Your Codebase is Old Enough to Vote
- ❑ Sequencing, Linkage, Inheritance
- ❑ Evolution of Graphics Technology
- ❑ GPU, Pipeline, and the Vector Graphics API
- ❑ Declarations and Type Conversions
- ❑ C++ ISO Standard
- ❑ Inline Namespaces
- ❑ Lambdas in Action
- ❑ Any Optional
- ❑ Variant
- ❑ CsPaint Library
- ❑ Moving to C++17
- ❑ What is the C++ Standard Library
- ❑ Attributes
- ❑ Copy Elision
- ❑ Time Complexity
- ❑ Qualifiers

Please subscribe to our YouTube Channel
https://www.youtube.com/copperspice

Libraries

- CopperSpice
  - libraries for developing GUI applications
- CsPaint Library
  - standalone C++ library for rendering graphics on the GPU
- CsSignal Library
  - standalone thread aware signal/slot library
- CsString Library
  - standalone unicode aware string library
- CsLibGuarded
  - standalone multithreading library for shared data

Applications

- KitchenSink
  - contains 30 demos and links with almost every CopperSpice library
- Diamond
  - programmers editor which uses the CopperSpice libraries
- DoxyPress & DoxyPressApp
  - application for generating source code and API documentation

Where to find CopperSpice

- www.copperspice.com
- twitter: @copperspice_cpp
- ansel@copperspice.com
- barbara@copperspice.com
- source, binaries, documentation files
  - download.copperspice.com
- source code repository
  - github.com/copperspice
- discussion
  - forum.copperspice.com