# ACCU December 2020

- Who we are
  - Barbara Geller
  - Ansel Sermersheim

- Co-Founders of CopperSpice
  - set of cross platform GUI libraries

- Co-Founders of DoxyPress
  - application to generate documentation

www.copperspice.com
twitter: @copperspice_cpp

# Lambda Expressions

- Overview
  - 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
    - fundamental definition in C++
      - an expression which returns a function object
    - an expression is something which returns a value, such as 5 + 2
    - the word “expression” is required here since evaluating a lambda expression actually returns a function object
    - when the function object is “invoked”, this produces some return value

# Lambda Expressions

- Terminology
  - 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

- Example
  - prior to C++11 this was the only way to create a function object
  - 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

- 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, must have the [ ]
  - parameter list
    - can be empty or omitted
  - return type
    - data type returned by the body is optional, normally deduced
  - body
    - contains the programming statements to execute
    - can be empty, must have the { }

# Lambda Expressions

- Capture Clause
  - generalized capture, added in C++14
    - 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
    - this->foo in the body refers to the original object
  - C++14
    - [self = *this]
    - capture *this object by value, initializes a new variable
  - C++17
    - [*this]
    - capture *this object by value
    - this->foo in the body refers to a copy of the object

# Lambda Expressions

- Full Syntax as of C++20
  - template parameters
    - added in C++20
    - same syntax used with a template function or method
  - defined to be equivalent
    - `(auto && arg)`
    - `<typename T> (T && arg)`

```
[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

- 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]

  - officially resolved in C++20
  - gcc and MSVC both allowed this capture pre C++20
  - known issue as of clang 11, still reports an error and it should not

# Concepts

- Definition of a Constraint
  - evaluated at compile time
  - applies a limitation on template parameters

  - T and R
    - placeholder or proxy for a data type
    - only types used in the given application are substituted
    - constraining T or R means the data type must satisfy the constraint, for the template to be valid

```
template <typename T, typename R>
R someFunction(T data);
```

# Concepts

- Definition of a Constraint
  - requires
    - keyword used to define a constraint
    - starts a requires clause
    - requires clause must be a constant expression which can be fully evaluated to a bool at compile time

  - concept
    - keyword used to associate a constraint with a name
    - roughly 30 built in concepts are provided in C++20
      - std::copyable<T>, std::derived_from<D, B>, std::swappable<T>
      - std::invocable<T, Args…>, std::equality_comparable_with<T, U>

# Concepts

- Requires vs Concept

```
template <typename T, typename R>                    // example 1
requires (sizeof(T) == sizeof(R))
R someFunction1(T data);
```

---

```
template <typename A, typename B>                    // example 2
concept SameSize = (sizeof(A) == sizeof(B));

template <typename T, typename R>
requires SameSize<T, R>
R someFunction2(T data);
```

# Concepts

- Why Concepts
  - SFINAE ( substitution failure is not an error )
    - technique used to eliminate a template from consideration
    - entities like decltype, enable_if, and type traits, are used so the compiler will not instantiate the given template
    - resolves overloaded templates based on the data type
  - concepts
    - can replace most uses of SFINAE
    - when template overload resolution results in ambiguity
      - template constraints will be considered and often this will resolve the conflict

# Concepts

- SFINAE vs Constraints

```
// T must be a floating point value                declaration A
template <typename T,
          typename X = std::enable_if_t<std::is_floating_point_v<T>> >
void someThing(T data);


// T must be a pointer value                       declaration B
template <typename T, typename = void,
          typename X = std::enable_if_t<std::is_pointer_v<T>> >
void someThing(T data);
```

# ACCU December 2020

- ❑ Constexpr Static Const
- ❑ C++ ISO Standard
- ❑ Any Optional
- ❑ Variant
- ❑ Moving to C++17
- ❑ What is the C++ Standard Library
- ❑ Attributes
- ❑ Copy Elision
- ❑ Time Complexity
- ❑ Qualifiers
- ❑ Concepts
- ❑ What is Initialization

https://www.youtube.com/copperspice

# Concepts

- SFINAE vs Constraints

```
// T uses a concept                                    declaration C
template <typename T>
requires std::floating_point<T>
void someThing(T data);


// T uses a type trait as a constraint                 declaration D
template <typename T>
requires (std::is_pointer_v<T>)
void someThing(T data);
```