# Lambda Expressions

This topic introduces Lambda Expressions which enable you to define anonymous function objects where they're passed to a function

- Before you can pass a function pointer or function object to an algorithm, the corresponding function or class must have been declared
- C++11's Lambda expressions (or lambda functions) enable you to define anonymous function objects where they're passed to a function
- They're defined locally inside functions and can "capture" (by value or by reference) the local variables of the enclosing function then manipulate these variables in the lambda's body

- Figure 16.16 demonstrates a simple lambda expression example that doubles the value of each element in an `int` array

```
const size_t SIZE = 4;
array< int, SIZE > values = { 1, 2, 3, 4 };
```

- Declares and initialize a small array of `int`s named `values`

```
for_each( values.cbegin(), values.cend(),
   []( int i ) { cout << i * 2 << endl; } );
```

- Calls the `for_each` algorithm on the elements of values
- The third argument to `for_each` is a *lambda expression*
- Lambdas begin with *lambda introducer* (`[]`)
  - Followed by a parameter list and function body

- Return types can be inferred automatically if the body is a single statement of the form `return` *expression*;
  - Otherwise, the return type is `void` by default or you can explicitly use a *trailing return type*
- The compiler converts the lambda expression into a function object
  - The lambda expression receives an `int`, multiplies it by 2 and displays the result.
- The `for_each` algorithm passes each element of the array to the lambda

```
int sum = 0;
for_each( values.cbegin(), values.cend(),
    [ &sum ]( int i ) { sum += i; } );
```

- Calculates the sum of the `array` elements
- The lambda introducer `[&sum]` indicates that this lambda expression *captures* the local variable `sum` *by reference*
  - So that the lambda can modify `sum`'s value
- Without the ampersand
  - `sum` would be captured by `value`
  - and the local variable outside the lambda expression would *not* be updated
- The `for_each` algorithm passes each element of values to the lambda
  - Which adds the value to the `sum`

- You can assign lambda expressions to variables
  - Which can then be used to invoke the lambda expression or pass it to other functions
- For example, you can assign the lambda expression to a variable as follows:

```
auto myLambda =
   []( int i )
      { cout << i * 2 << endl; };
```

- You can then use the variable name as a function name to invoke the lambda as in:

```
myLambda( 10 ); // outputs 20
```

This topic introduced Lambda Expressions which enable you to define anonymous function objects where they're passed to a function