Function Objects

This topic teaches Function Objects which are implemented as an object of a class that overloads the function-call operator (parentheses) with a function named operator()

- Many Standard Library algorithms allow you to pass a function pointer into the algorithm to help the algorithm perform its task
  - `binary_search` is overloaded with a version that requires as its fourth parameter a *function pointer* that takes two arguments and returns a `bool` value
  - The algorithm uses this function to compare the search key to an element in the collection
  - The function returns
    - `true` if the search key and element being compared are equal
    - Otherwise, the function returns `false`

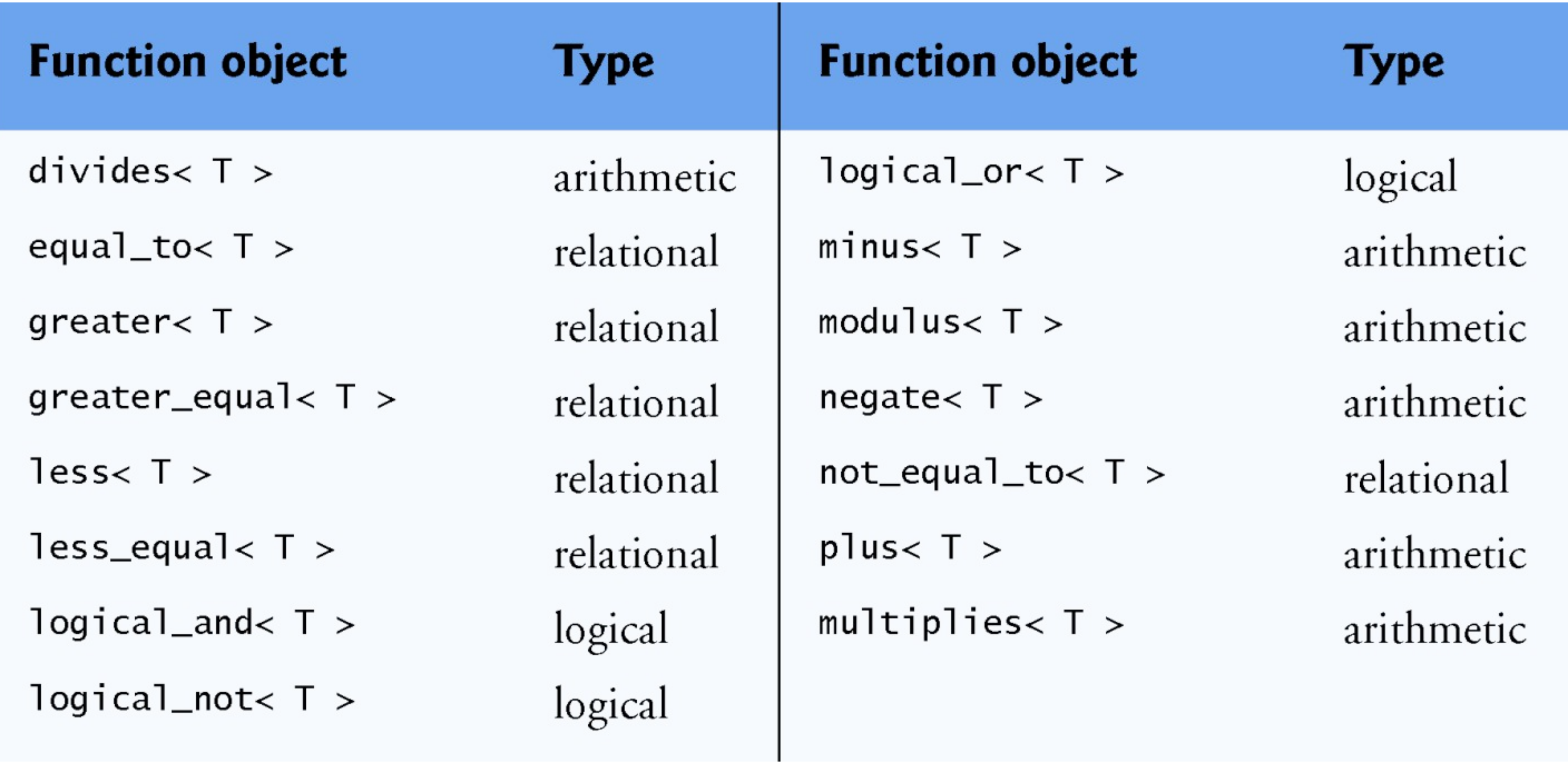
- This enables `binary_search` to search a collection of elements for which the element type does *not* provide an overloaded equality < operator
- Any algorithm that can receive a *function pointer*
  - Can also receive an object of a class that overloads the function-call operator (parentheses) with a function named `operator()`
  - Provided that the overloaded operator meets the requirements of the algorithm
    - in the case of `binary_search`, it must receive two arguments and return a `bool`

- An object of such a class is known as a function object
  - can be used syntactically and semantically like a function or *function pointer*
  - The overloaded parentheses operator is invoked by using a function object's name followed by parentheses containing the arguments to the function
- Most algorithms can use function objects and functions interchangeably

Function objects provide several advantages over *function pointers*.

- The compiler can inline a *function object's* overloaded `operator()` to improve performance
- Also, since they're objects of classes, *function objects* can have data members that `operator()` can use to perform its task

Many predefined function objects can be found in the header <functional>



Function object	Type
divides< T >	arithmetic
equal_to< T >	relational
greater< T >	relational
greater_equal< T >	relational
less< T >	relational
less_equal< T >	relational
logical_and< T >	logical
logical_not< T >	logical
logical_or< T >	logical
minus< T >	arithmetic
modulus< T >	arithmetic
negate< T >	arithmetic
not_equal_to< T >	relational
plus< T >	arithmetic
multiplies< T >	arithmetic

Figure 16.15 uses the `accumulate` numeric algorithm (introduced in Fig. 16.30) to calculate the sum of the squares of the elements in an `array`.

The fourth argument to `accumulate` is a binary function object (that is, a *function object* for which `operator()` takes two arguments) or a function pointer to a binary function (that is, a function that takes two arguments)

Function `accumulate` is demonstrated twice—once with a *function pointer* and once with a *function object*

```
int sumSquares( int total, int value )
{
   return total + value * value;
} // end function sumSquares
```

- Defines a function `sumSquares` that squares its second argument `value`, adds that square and its first argument `total` and returns the sum
- Function `accumulate` will pass each of the elements of the sequence over which it iterates as the second argument to `sumSquares` in the example
- On the first call to `sumSquares`, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`; `0` in this program)
- All subsequent calls to `sumSquares` receive as the first argument the running sum returned by the previous call to `sumSquares`
- When `accumulate` completes, it returns the sum of the squares of all the elements in the sequence

```
template< typename T >
class SumSquaresClass
{
public:
   T operator()(const T &total,
                const T &value)
   {
      return total + value * value;
   }
};
```

- Defines `SumSquaresClass` with an overloaded `operator()` that has two parameters and returns a value—the requirements for a binary function object
- On the first call to the *function object*, the first argument will be the initial value of the `total` (which is supplied as the third argument to accumulate; `0` in this program) and the second argument will be the first element in `array integers`
- All subsequent calls to `operator` receive as the first argument the result returned by the previous call to the *function object*, and the second argument will be the next element in the `array`
- When `accumulate` completes, it returns the sum of the squares of all the elements in the `array`

```
int result = accumulate
   (integers.cbegin(), integers.cend(),
    0, sumSquares);
```

- Calls function `accumulate` with a *pointer to function* `sumSquares` as its last argument

```
result = accumulate
   (integers.cbegin(), integers.cend(),
    0, SumSquaresClass< int >() );
```

- Calls `accumulate` with an object of class `SumSquaresClass` as the last argument

- The expression `SumSquaresClass< int >()`
  - Creates (and calls the default constructor for) an instance of class `SumSquaresClass` (a *function object*)
  - That is passed to `accumulate`, which invokes function `operator()`

```
result = accumulate
    (integers.cbegin(), integers.cend(),
     0, SumSquaresClass< int >()
```

- Could be written as two separate statements

```
SumSquaresClass< int > sumSquaresObject;
result = accumulate( integers.cbegin(),
   integers.cend(), 0, sumSquaresObject );
```

- The first line defines an object of class `SumSquaresClass`.
- That object is then passed to function `accumulate`.

This topic taught Function Objects which are implemented as an object of a class that overloads the function-call operator (parentheses) with a function named operator()