```
#include <algorithm>  // std::count_if
#include <numeric>    // std::accumulate

std::size_t numAboveAverageSalaries(const std::vector<Employee>& employees)
{
    const ~~long~~ sum = std::accumulate(employees.begin(), employees.end(), 0L,
                                     SalaryAccumulator());

    const ~~long~~ average = sum / employees.size();
    return std::count_if(employees.begin(), employees.end(),
                         SalaryIsGreater(average));
}
```

We now turn our attention to a syntax that allows us to rewrite these examples much more simply and compactly. Returning to the sorting example, the rewritten code has the name-comparison and salary-comparison operations expressed in place, within the call to `std::sort`:

```
void sortByName2(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(),
              [](const Employee& e1, const Employee& e2)
              {
                  return e1.name < e2.name;
              });
}

void sortBySalary2(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(),
              [](const Employee& e1, const Employee& e2)
              {
                  return e1.salary < e2.salary;
              });
}
```

In each case, the third argument to `std::sort` — beginning with `[]` and ending with the nearest closing `}` — is called a **lambda expression**. Intuitively, for this case, one can think of a lambda expression as an *operation* that can be invoked as a callback by the algorithm. The example shows a function-style parameter list — matching that expected by the `std::sort` algorithm — and a function-like body that computes the needed predicate. Using lambda expressions, a developer can express a desired operation directly at the point of use rather than defining it elsewhere in the program.

The compactness and simplicity afforded by using lambda expressions is even more evident when we rewrite the average-salaries example:

```
std::size_t numAboveAverageSalaries2(const std::vector<Employee>& employees)
```

```
{
    if (employees.empty()) { return 0; }
    const long sum = std::accumulate(employees.begin(), employees.end(), 0L,
                                     [](long currSum, const Employee& e)
                                     {
                                         return currSum + e.salary;
                                     });

    const long average = sum / employees.size();
    return std::count_if(employees.begin(), employees.end(),
                         [average](const Employee& e)
                         {
                             return e.salary > average;
                         });
}
```

The first lambda expression, above, specifies the operation for adding another salary to a running sum. The second lambda expression returns true if the `Employee` argument, `e`, has a salary that is larger than `average`, which is a local variable *captured* by the lambda expression. A **lambda capture** is a set of local variables that are usable within the body of the lambda expression, effectively making the lambda expression an extension of the immediate environment. We will look at the syntax and semantics of lambda captures in more detail in the next section, *Parts of a lambda expression*, below.

Note that the lambda expressions replaced a significant portion of code that was previously expressed as separate functions or functor classes. Some of that code reduction is in the form of documentation (comments), which increases the appeal of lambda expressions to a surprising degree. Creating a named entity such as a function or class imposes on the developer the responsibility to give that entity a meaningful name and sufficient documentation for a future human reader to understand its *abstract* purpose, outside the context of its use, even for one-off, nonreusable entities. Conversely, when an entity is defined right at the point of use, it might not need a name at all, and it is often self-documenting, as in both the sorting and average-salaries examples above. Both the original creation and maintenance of the code are simplified.

Parts of a lambda expression

A lambda expression has a number of parts and subparts, many of which are optional. For exposition purposes, let’s look at a sample lambda expression that contains all of the parts:

```
     introducer
   ______^______
  /             \
 [&, cap2, cap3](T1 arg1, T2 arg2) mutable noexcept -> ReturnType { /*...*/ }
  \_____ _____/ \___________________ ___________________/ \______ ______/
        V                           V                            V
     capture                    declarator                      body
```