Anonymous Function Objects (Closures)

Lambda expressions provide a means of defining function objects at the point where they are needed, enabling a powerful and convenient way to specify callbacks or local functions.

Description

Generic, object-oriented, and functional programming paradigms all place great importance on the ability of a programmer to specify a *callback* that is passed as an argument to a function. For example, the Standard Library algorithm, `std::sort`, accepts a callback argument specifying the sort order:

```
#include <algorithm>   // std::sort
#include <functional>  // std::greater
#include <vector>      // std::vector

template <typename T>
void sortDescending(std::vector<T>& v)
{
    std::sort(v.begin(), v.end(), std::greater<T>());
}
```

The function object, `std::greater<T>()`, is callable with two arguments of type `T` and returns **`true`** if the first is greater than the second and **`false`** otherwise. The Standard Library provides a small number of similar **functor types**, but, for more complicated cases, programmers must write a **functor** themselves. If a container holds a sequence of `Employee` records, for example, we might want to sort the container by either name or salary:

```
#include <string>  // std::string
#include <vector>  // std::vector

struct Employee
{
    std::string name;
    long        salary;  // in whole dollars
};

void sortByName(std::vector<Employee>& employees);
void sortBySalary(std::vector<Employee>& employees);
```

The implementation of `sortByName` can delegate the sorting task to the standard algorithm, `std::sort`. However, to achieve sorting by the desired criterion, we will need to supply `std::sort` with a callback that compares the names of two `Employee` objects. We can implement this callback as a pointer to a simple function that we pass to `std::sort`: