Function Objects and Lambda Functions

Function objects

- Function objects, also known as functors, provide an alternative to the `RealFunction` class. Recall we introduced this to write an integration routine.
- They allow you to use lambda functions, a powerful technique to write classes more quickly

Integration using function objects

```
double integrate(
                  function<double(double)> f,
                  double a,
                  double b,
                  int nSteps) {
    double total = 0.0;
    double h = (b - a) / nSteps;
    for (int i = 0; i<nSteps; i++) {
        double x = a + i*h + 0.5*h;
        total += h*f(x);
    }
    return total;
}
```

Writing a function object

```
class SinFunction {
public:
    double operator()( double x) {
        return sin(x);
    }
};
```

Testing it

```
void testIntegrateSin() {
    SinFunction integrand;
    double value = integrate(integrand, 0, 1, 1000);
    ASSERT_APPROX_EQUAL(-cos(1.0) + cos(0.0),
        value, 0.01);
}
```

Lambda functions

- Suppose we want to write a function to calculate

$$\int_0^1 (ax^2 + bx + c)\,\mathrm{d}x$$

```
class QuadraticFunction {
public:
    /*  Members */
    double a;
    double b;
    double c;
    /*  Constructor */
    QuadraticFunction(double a,
        double b,
        double c) :
        a(a), b(b), c(c) {}
    /*  Operator */
    double operator()(double x) {
        return a*x*x+b*x+c;
    }
};

double integrateQuadratic(double a,
    double b,
    double c) {
    QuadraticFunction integrand(a, b, c);
    return integrate(integrand, 0, 1, 1000);
}
```

With a lambda function

```
double integrateQuadratic2(double a,
    double b,
    double c) {
    auto lambda =
        [a, b, c](double x) {
        return a*x*x + b*x + c;
    };
    return integrate(lambda, 0, 1, 1000);
}
```

- Generate a class, we don't care what it is called.
- We will want to *capture* the local variables `a`, `b` and `c` of `integrateQuadratic2` and have them as member variables of our class. When writing a lambda function, you list the captured variables in square brackets.
- We want to write an overload of operator `()` that takes a single `double` parameter which we will call `x`. When writing a lambda function, you list the parameter types and parameter names in round brackets.
- The actual computation for the function is written inside curly brackets and can use both the captured variables and the parameters.

In summary, the syntax of a lambda function is:

```
[CaptureParameters] (FunctionParameters) {
    FunctionImplementation
}
```

There is a lot of flexibility in how you write the capture parameters.

(i) You can specify that you would like to capture local variables by reference, by using the `&` symbol before the parameter name.

(ii) You can specify that you would like to capture all variables by reference simply by specifying just `&`.

(iii) You can specify that you would like to capture all variables by value, by specifying just `=`.

(iv) If your lambda function is written inside a member function of a class, you can capture the member functions and member variables of that class by specifying `this`.

Integrating an option's payoff

```
double integratePayoff(PathIndependentOption& o,
                       double a,
                       double b) {
    auto lambda =
        [&o](double x) {
        return o.payoff(x);
    };
    return integrate(lambda, a, b, 1000);
}
```

Function pointers

To compute

$$\int_0^1 (x^2 + 2x + 1)\mathrm{d}x$$

we can write an ordinary function representing the integrand as follows:

```
static double integrand(double x) {
    return x*x + 2 * x + x;
}
```

Function pointers

We can then pass this integrand to our integrate function

```
double testIntegrateFunctionPointer() {
    double value = integrate(&integrand, 2, 1);
    ASSERT_APPROX_EQUAL(
        value,
        2.3333, 0.01);
}
```

- Note the & symbol before `integrand`.
- `&integrand` is called a *function pointer*.

Sorting with lambda functions

```
void sortExample() {
    vector<string> list({ "Z", "x", "a", "B" });
    sort(list.begin(), list.end(),
        [](string& x, string& y) {
            return uppercase(x) < uppercase(y);
    });
}
```

Summary

- Passing functions as parameters is a common requirement in C++.
- Use the class `std::function` to pass functions as parameters.
- Use lambda functions to quickly write new function objects.