Function things

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

May 30, 2017

Overview I

Functors

- Is an object that acts like a function.
- Provides plug-in that adapts the behaviour of other methods/objects.
- E.g Print all elements in an array that have some property:
  - Problem: Don't know what kind of properties.
  - Solution: Supply an object as a parameter that has:
    - A method to determine whether an element has a property.
    - Instance variables that describe the particular property.

```
template<typename T, int N, typename F>
void display(T (&array)[N], F& hasProperty) {
        for (int i = 0; i < N; i++) {
                if (hasProperty(array[i]))
                        cout << array[i] << " ";
        }
}
```

Example

Determine if a number is a fixed multiple of some integer:

```
template <typename T>
class IsMultipleOf {
private:
        T n;
public:
        IsMultipleOf(T n){ this->n = n; }
// Use object like a function
        bool operator()(T& v){
                return v%n == 0;
        }
}
int main(int argc, char* argv) {
        int array[] =
            {+2,-6,+3,-5,+7,-9,+3,-2,+4,-5,+8,-8,+9};
        IsMultipleOf<int> isMultiple(3);
// Prints all multiples of 3
        display(array, isMultiple);
        return 0;
}
```

Lambdas

- Basically an anonymous function
- Alternative to functors.

```
[N](int n) const char* {return (n%N==0)?"even":"odd"}
```

- Have four parts:
  - Capture list: `[N]`
  - Parameter list: `(int n)`
  - Optional return type, compiler will deduce one if not specified: `const char*`
  - The body: `{return (n%N==0)?"even":"odd"}`
- Can capture local and global variables of the scope in which they are declared.

Example

- Passed to templates the same way as Functors:

```
int main(int argc, char* argv) {
        int array[] =
            {+2,-6,+3,-5,+7,-9,+3,-2,+4,-5,+8,-8,+9};

// Prints all even numbers
        display(array, [](int n){return !(n%2);});
        return 0;
}
```

Capture by value

- Simplest/Safest way is to capture by value, therefore a copy is passed to the lambda.

```
int main(int argc, char* argv) {
        int array[] =
            {+2,-6,+3,-5,+7,-9,+3,-2,+4,-5,+8,-8,+9};
        int N = 3;

// Prints all multiples of 3, captured by value
        display(array, [N](int n){return !(n%N);});
        return 0;
}
```

Capture by reference

- Lambda can in turn, alter the value taken in by reference using &.

```
int main(int argc, char* argv) {
        int array[] =
            {+2,-6,+3,-5,+7,-9,+3,-2,+4,-5,+8,-8,+9};
        int N = 1;

// Prints all multiples of N where n doubles each iteration?
        display(array, [&N](int n){return !(n%N*2);});
// N is 2,4,8,16.... etc..
        return 0;
}
```

- Default is capture by value, unless specified by &. Can use:
  - [&, *list...*], to capture all by reference unless otherwise specified.

Storing lambdas

- Can use `auto`, leaves to the compiler to deduce type.

```
auto multipleOfN = [N](int n){if (n%N == 0) cout << n;};
```

- Alternatively can use `std::function<R(AL)>`
  - R is the return type.
  - AL is comma seperated list of arguments

```
std::function<boolean(int)> multipleOfN = [N](int n) boolean
    {if (n%N == 0) cout << n;};
```

The End