Functions and Function Objects

Regular functions and function objects (objects of a class that overloads the function call operator) are equivalent — they are called in the same way and may both be passed as arguments to other functions. Function objects (sometimes called functors) may have state and are often more efficient to use than regular functions. This is because calls of a function object can be inlined.

Algorithms and Functions

Functions and function objects are used to parameterize an algorithm with an "operation". The library function `for_each` traverses a range and performs an operation on all elements:

```
template <typename It, typename Op>
  // It is an InputIterator
  // Op is a function with one parameter
Op for_each(It beg, It end, Op op) {
    for (; beg != end; ++beg) {
        op(*beg);
    }
    return op;
}
```

The argument op may be a function or a function object, as long as it has one parameter.

for_each, Regular Functions

for_each can be used with regular functions:

```
void clear(int& value) { value = 0; }

void print(int value) { cout << value << " "; }

int main() {
    vector<int> v = {10, 20, 30, 40, 50};

    for_each(v.begin(), v.end(), print);
    cout << endl;

    for_each(v.begin(), v.end(), clear);

    for_each(v.begin(), v.end(), print);
    cout << endl;
}
```

for_each, Function Objects

On slide 171 we showed an example of a function object with state. The same class, now as a class template, and an example of use:

```
template <typename T>
class Accumulator {
public:
    Accumulator() : sum() {}
    T get_sum() const { return sum; }
    void operator()(const T& x) { sum += x; }
private:
    T sum;
};

vector<int> v = {10, 20, 30, 40, 50};
Accumulator<int> accum;
accum = for_each(v.begin(), v.end(), accum);
cout << accum.get_sum() << endl;
```

Function Examples — Finding

The library algorithm `find_if` returns an iterator to the first element for which the function `pred` returns true.

```
template <typename InIt, typename Pred>
InIt find_if(InIt beg, InIt end, Pred pred) {
    while (beg != end && !pred(*beg)) {
        ++beg;
    }
    return beg;
}
```

We will use `find_if` to find an element smaller than a given value:

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(), ???) != v.end();
}
```

Finding With a Regular Function

A regular function can be used, but the searched-for value must be stored in a global variable. This is not an attractive solution.

```
int global_value;

bool is_less(int x) { return x < global_value; }

bool exists_smaller(const vector<int>& v, int value) {
    global_value = value;
    return find_if(v.begin(), v.end(), is_less) != v.end();
}
```

Finding With a Function Object

A solution with a function object is much more flexible:

```
class Less {
public:
    Less(int v) : value(v) {}
    bool operator()(int x) const { return x < value; }
private:
    int value;
};

bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(), Less(value)) != v.end();
}
```

Also, this solution is more efficient than the previous one. The call to the comparison function can be inlined by the compiler. This is not possible when the argument is a function pointer.

Finding With a Lambda C++11

In the new standard, the definition of a function object can be written at the place of the call. This is called a lambda function.

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(),
                   [value](int x) { return x < value; }) != v.end();
}
```

`[value]` access the variable `value` in the current scope. Variables can also be captured by reference: `[&value]`

`(int x)` normal parameter list

`) {` the return type is deduced from the `return` statement. Can be specified as `-> bool`

`{ ... }` function body

The Compiler Generates a Class From a Lambda

When you define a lambda, the compiler writes a function class and creates a function object:

```
auto less = [value](int x) -> bool { return x < value; }
```

This is what the compiler generates (compare with slide 244):

```
class Less {
public:
    Less(int v) : value(v) {}
    bool operator()(int x) const { return x < value; }
private:
    int value;
};

auto less = Less(value);
```

Library-Defined Function Objects

The algorithm `sort(beg,end)` sorts an iterator range in ascending order (values are compared with <). A function defining the sort order can be supplied as a third argument. To sort in descending order (comparing values with >):

```
vector<string> v;
...
sort(v.begin(), v.end(),
        [](const string &a, const string& b) { return a > b; });
```

Simple function objects like `greater`, `less`, `equal_to` are available in the library. They can be used like this:

```
sort(v.begin(), v.end(), greater<string>());
```

Binding Function Arguments C++11

The library-defined function objects can often be used together with the library algorithms, but sometimes they don't exactly match. We return to the problem of finding an element less than a given value (slide 242).

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(), ???) != v.end();
}
```

`find_if` takes each element from the vector and checks if the function returns true. Here, each element should be checked against `value`. The library function object `less` takes two parameters, and we must *bind* the second argument to `value`:

```
bool exists_smaller(const vector<int>& v, int value) {
    return find_if(v.begin(), v.end(),
                   bind(less<int>(), _1, value) != v.end();
}
```

Algorithms

There are about 100 algorithms in the header `<algorithm>`. They can be classified as follows:

Nonmodifying look at the input but don't change the values or the order between elements (`find`, `find_if`, `count`, ...).

Modifying change values, or create copies of elements with changed values (`for_each`, `copy`, `fill`, ...).

Removing remove elements (`remove`, `unique`, ...).

Mutating change the order between elements but not their values (`reverse`, `random_shuffle`, ...).

Sorting a special kind of mutating algorithms (`sort`, ...).

Sorted Range require that the input is sorted (`binary_search`, ...).

Numeric for numeric processing (`accumulate`, `inner_product`, ...).

Algorithm Notes

- The input to an algorithm is one or more iterator ranges `[beg,end)`. If the algorithms writes elements, the start of the output range is given as an iterator.
- Return values are often iterators.
- Some algorithms are used for the same purpose and have the same number of arguments. They are not overloaded, instead they have different names. An example is `find` for finding an element with a given value, `find_if` for finding an element for which a predicate is true.
- Some containers have specialized algorithms that are more efficient than the library algorithms, for instance `map::find` for finding a value in a map (binary search tree).

Finding Algorithms

We have already seen examples of algorithms that find values. These algorithms return the end iterator if the value isn't found. Other examples:

```
vector<string> v = ...;

cout << "There are " << count(v.begin(), v.end(), "hello")
     << " hello's in v" << endl;

vector<string> v2 = {"hello", "hej", "davs"};
auto it = find_first_of(v.begin(), v.end(), v2.begin(), v2.end());
```

Modifying Algorithms

The following examples use strings — strings have iterators that can be used with the library algorithms.

```
string s = "This is a sentence";

// convert the string to lower case. The C function
// tolower(ch) returns ch in lower case
transform(s.begin(), s.end(), s.begin(), ::tolower);

// print all non-blanks
copy_if(s.begin(), s.end(), ostream_iterator<char>(cout),
        [](char c) { return c != ' '; });

// fill s with blanks
fill(s.begin(), s.end(), ' ');
```

Removing Algorithms

The most important fact about the removing algorithms is that they don't actually remove elements from a container (since the algorithms have iterators as input they don't know which container to remove from, or how). Instead they reorder the elements based on a criterion.

```
vector<int> v = {1, 3, 2, 6, 3, 3, 4};

auto it = remove(v.begin(), v.end(), 3);
// v contains {1, 2, 6, 4, ...}, it points after the 4
// vector::erase really removes elements:
v.erase(it, v.end());

v = {1, 3, 3, 5, 7, 7, 7, 8, 9, 9};
v.erase(unique(v.begin(), v.end()), v.end());
```

Mutating and Sorting Algorithms

```
vector<int> v = {1, 4, 44, 3, 15, 6, 5, 7, 7, 4, 22, 1};
// reorder so odd numbers precede even numbers
partition(v.begin(), v.end(), [](int x) { return x % 2 != 0; });

// shuffle the elements (requires random access iterators)
random_shuffle(v.begin(), v.end());

// sort until the first 5 elements are correct
partial_sort(v.begin(), v.begin() + 5, v.end());
```

Sorted Range Algorithms

These algorithms require that the input range is sorted. Example (insert unique numbers in a sorted vector):

```
int main() {
    vector<int> v;
    int nbr;
    while (cin >> nbr) {
        auto it = lower_bound(v.begin(), v.end(), nbr);
        if (it == v.end() || *it != nbr) {
            v.insert(it, nbr);
        }
    }
    ...
}
```

Algorithm Example

Read words from standard input, remove duplicate words, print in sorted order:

```
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
#include <iostream>

int main() {
    using namespace std;
    vector<string> v((istream_iterator<string>(cin)),
                     istream_iterator<string>());
    sort(v.begin(), v.end());
    unique_copy(v.begin(), v.end(),
                ostream_iterator<string>(cout, "\n"));
}
```

Algorithm Summary

- Every C++ programmer should be familiar with the algorithms.
- The algorithms are easy to use, well tested and efficient.
- We have only shown a few of the algorithms. Look in a book or at documentation on the web, for example at
  - `http://en.cppreference.com/w/cpp/algorithm`, or
  - `http://www.cplusplus.com/reference/algorithm`