Functions, Function objects, and Function Pointers

It is possible to create pointers to functions and create object instances that act like functions.

Function pointers can be passed to functions and put in arrays etc.

All function types are coerced to pointers to functions when used as values.
Example Using Function Pointers

```cpp
#include <iostream>
using namespace std;
int same( int i ) {
    return i;
}
int square( int i ) {
    return i*i;
}
int cube( int i ) {
    return i*i*i;
}
typedef int (*IntFunct)(int) ;
int main( int argc, char *argv[] ) {
    IntFunct functarray[3] = {same, square, cube};
    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 3; j++) {
            cout << functarray[j](i) << " ";
        }
        cout << endl;
    }
}

swm[1479] % a.out
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
Writing a summation function

Here is a function that sums an argument function between given limits

```c++
int same( int i ) {
    return i;
}

int square( int i ) {
    return i*i;
}

int cube( int i ) {
    return i*i*i;
}

typedef int (*IntFunct)(int);

int sum(IntFunct f, int low, int high) {
    int sum = 0;
    for(int i = low; i <= high; i++) {
        sum += f(i);
    }
    return sum;
}

int main( int argc, char *argv[] ) {
    cout << "Sum from 1 to 10 is: " << sum(same, 1, 10) << endl;
    cout << "Sum squares from 1 to 10 is: " << sum(square, 1, 10) << endl;
    cout << "Sum cubes from 1 to 10 is: " << sum(cube, 1, 10) << endl;
}
```

Sum from 1 to 10 is: 55
Sum squares from 1 to 10 is: 385
Sum cubes from 1 to 10 is: 3025
Function Pointers

Functions can only be declared at the top level in C++
Cannot have state that survives from call to call without using global variables (ugly)

Cannot have a function that returns created functions
  Cannot have a function that makes functions that add a specified integer to its argument

Function objects are more flexible
Function Objects

Classes can define operator() which makes instances of the class behave like functions

class Adder {
    int i;
public:
    Adder(int toAdd) : i(toAdd) {}

    int operator()(int j) {
        return i+j;
    }
};

Adder a5(5); // a function object that adds 5
Adder a7(7); // a function object that adds 7

a5(9)  --->  returns 14
a7(4)  --->  returns 11

The type of a function object is the type of its class
Note that the type of a5 is "Adder" and this type cannot be converted to a function type
Function Objects and Templates

We cannot pass in these Adder objects to our sum function - Adder is not of function type

Adders have all of the characteristics of functions - so we can use templates

Define a templated function whose template parameter that matches the type of the functional argument

Use the argument as a function inside of the templated function

   If the argument is a function then a function will be called
   If the argument is a class with operator() defined then operator() will be called
Templated function with Function Objects
#include <iostream>

using namespace std;

int same( int i ) {
    return i;
}

class Adder {
    int i;
public:
    Adder(int toAdd) : i(toAdd) {}

    int operator()(int j) {
        return i+j;
    }
};

template<class Function>
int sum(Function f, int low, int high) {
    int sum = 0;
    for(int i = low; i <= high; i++) {
        sum += f(i);
    }
    return sum;
}

int main( int argc, char *argv[] ) {
    cout << "Sum from 1 to 10 is: " << sum(same, 1, 10) << endl;
    cout << "Sum i+5 from 1 to 10 is: " << sum(Adder(5), 1, 10) << endl;
}

Sum from 1 to 10 is: 55
Sum i+5 from 1 to 10 is: 105
Cautions

Functional objects are like other objects
They have a lifetime and cannot be used after being destroyed
Creation by new must have a corresponding delete

In the expression
sum(Adder(5), 1, 10)

The Adder object is created as a temporary and lasts only for the statement
In this example, the Adder object is not needed after the statement

Some functional objects take other functional objects in their constructor and the resulting object is valid only in a single statement

count_if (v.begin, v.end,
    bind1st(greater<int> (),7),
    littleNums)
Functional Objects and the STL

In the `<functional>` library are many utility functional objects. There is a functional object for every arithmetic operator (+, -, * etc.) and logical operator (and, or, not).

There are functional objects for composing functions, fixing one argument of a binary operator.

You can write your own objects that will work with the standard template library.
Example of functional objects

If a C++ program wants to have a by-element addition of two vectors a and b containing double and put the result into a, it can do:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

`transform (in this version) operates like the code:
```
transform(start1, end1, start2, result, function) {
    for(iter1 = start1, iter2 = start2;
         iter1 != end1;
         ++iter1, ++iter2, ++result) {
        *result = function(*iter1, *iter2);
    }
    return result;
}
```
Some algorithms

The `<algorithm>` library has many uses of functional objects

Comparison objects for sorts and searches

Calculation objects for transformations

Types of Algorithms

Non-mutating

accumulate, adjacent_find, binary_search, count_min, count_if, equal,

non-mutating

equal_range, find, find_end, find_first_of, find_if, for_each, includes,

lexicographical_compare, lower_bound, max, max_element, min,

min_element, mismatch, nth_element, search, search_n,

Mutating

copy, copy_backward, fill, fill_n, generate, generate_n, inplace_merge,

iter_swap, make_heap, merge, nth_element, next_permutation,

partial_sort, partial_sort_copy, partition, prev_permutation, push_heap,

pop_heap, random_shuffle, remove, remove_copy, remove_copy_if,

remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse,

reverse_copy, rotate, rotate_copy, set_difference,

set_symmetric_difference, set_intersection, set_union, sort, sort_heap,

stable_partition, stable_sort, swap, swap_ranges, transform, unique,

unique_copy
Summary

With careful design some very high level constructs can be written

Use of the STL can result in very efficient code

  The algorithms in the STL are probably better than most people can write

Really neat code can be written using functional objects but care must be exercised

  Best way is to look at other examples of STL usage