C family evolution:
From function pointer to lambda function

Function Pointers

Function pointers in C

- Why they are necessary?
  - Functions as argument to other functions
    - Like when you want to tell the sort function how to behave
  - Callback functions/Listeners
    - Like when you are implementing a low level library and you want to let the higher level application implement its own function and lower level library use higher level application function

Function pointer syntax

- Return type (* function pointer name) (input types)
  - void * (*foo) (char *, int)

Initialize and using function pointer

```
#include <stdio.h>
void my_int_func(int x)
{
    printf( "%d\n", x );
}



int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    /* call my_int_func (note that you do not need to write (*foo)(2) ) */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );

    return 0;
}
```

A sort example

```
#include <stdlib.h>

int int_sorter( const void *first_arg, const void *second_arg )
{
    int first = *(int*)first_arg;
    int second = *(int*)second_arg;
    if ( first < second )
    {
        return -1;
    }
    else if ( first == second )
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
```

```
int main()
{
    int array[10];
    int i;
    /* fill array */
    for ( i = 0; i < 10; ++i )
    {
        array[ i ] = 10 - i;
    }

    Int (*comp) (const void*, const void*);
    Comp = &int_sorter;
    qsort( array, 10 , sizeof( int ), int_sorter );
    // Second solution
    //qsort( array, 10 , sizeof( int ), comp );
    for ( i = 0; i < 10; ++i )
    {
        printf ( "%d\n" ,array[ i ] );
    }

}
```

Using Polymorphism and Virtual Functions Instead of Function Pointers in C++

```
class Sorter
{
    public:
    virtual int compare (const void *first,
                              const void *second) = 0;
};

// cpp_qsort, a qsort using C++ features
// like virtual functions
void cpp_qsort(void *base, size_t nmemb,
                    size_t size, Sorter *compar);
```

```
class AscendSorter : public Sorter
{

    virtual int compare (const void*, const void*)
    {
        int first = *(int*)first_arg;
        int second = *(int*)second_arg;
        if ( first < second )
        {
            return -1;
        }
        else if ( first == second )
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }
};
```

```
Sorter * asorter = new AscendSorter();
cpp_qsort(array, 10, sizeof(int), asorter);
```

Function Objects

Functors: Function Objects in C++

```
class Message
{
    public:
        std::string getHeader (const std::string& header_name) const;
        // other methods...
};

class MessageSorter
{
    public:
        // take the field to sort by in the constructor
        MessageSorter (const std::string& field) : _field( field ) {}
        bool operator ()(const Message& lhs, const Message& rhs)
        {
            // get the field to sort by and make the comparison
            return lhs.getHeader( _field ) < rhs.getHeader( _field );
        }
    private:
        std::string _field;
};
```

```
std::vector<Message> messages;
// read in messages
MessageSorter comparator;
sort( messages.begin(), messages.end(), comparator );
//Actually you are passing an object and not a function! Beautiful!
```

Advantage of Functor

- Actually you are passing an object and not a function.
- You could also ask then why to override the operator? We could use a function of that class.
- But actually functor makes C++ backward compatible.
- How?

Functors, Function Pointers, and Templates

```
#include <string>
#include <vector>

class AddressBook
{
    public:
    // using a template allows us to ignore the differences between functors, function pointers
    // and lambda
    template<typename Func>
    std::vector<std::string> findMatchingAddresses (Func func)
    {
        std::vector<std::string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr )
        {
            // call the function passed into findMatchingAddresses and see if it matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<std::string> _addresses;
};
```

Functors vs. Virtual Functions

```
template <FuncType>
int doMath (int x, int y, FuncType func)
{
    return func( x, y );
}
```

```
class MathComputer
{
    virtual int computeResult (int x, int y) = 0;
};

doMath (int x, int y, MathComputer* p_computer)
{
    return p_computer->computeResult( x, y );
}
```

Lambda functions

Basic Lambda Syntax

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```

Lambda example

```
AddressBook global_address_book;

vector<string> findAddressesFromOrgs ()
{
    return global_address_book.findMatchingAddresses(
        // we're declaring a lambda here; the [] signals the start
        [] (const string& addr) { return addr.find( ".org" ) != string::npos; }
    );
}
```

Variable capture in lambda

```
// read in the name from a user, which we want to search
string name;
cin>> name;
return global_address_book.findMatchingAddresses(
    // notice that the lambda function uses the the variable 'name'
    [&] (const string& addr) { return addr.find( name ) != string::npos; }
);
```

Using lambda in for_each

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for ( auto itr = v.begin(), end = v.end(); itr != end; itr++ )
{
    cout << *itr;
}
```

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for_each( v.begin(), v.end(), [] (int val){ cout << val;} );
```

for_each has about the same performance, and is **sometimes even faster** than a regular for loop. (The reason: it can take advantage of loop unrolling.)

More on lambda

```
using namespace std;
#include <iostream>

int main()
{
    // You can directly call the lambda function.
    //Also you can remove the parameters if there is none.

    [] { cout << "Hello Wold!"; }();
}
```

Return value

```
[] () { return 1; } // compiler knows this returns an integer
```

```
[] () -> int { return 1; } // now we're telling the compiler what we want
```

Lambda closure

[]	Capture nothing (or, a scorched earth strategy?)
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable foo by reference
[bar]	Capture bar by making a copy; don't copy anything else
[this]	Capture the this pointer of the enclosing class

Lambda example

```
class Foo
{
public:
    Foo () : _x( 3 ) {}
    void func ()
    {
        // a very silly, but illustrative way of printing out the value of _x
        [this] () { cout << _x; } ();
    }

private:
        int _x;
};

int main()
{
    Foo f;
    f.func();
}
```

How are Lambda Closures Implemented?

How does the magic of variable capture really work?

It turns out that the way lambdas are implemented is by creating a small class; this class overloads the operator(), so that it acts just like a function.

A lambda function is an instance of this class; when the class is constructed, any variables in the surrounding environment are passed into the constructor of the lambda function class and saved as member variables.

What type is a Lambda?

C++11 does include a convenient wrapper for storing any kind of function--lambda function, functor, or function pointer: **std::function**.

```
#include <functional>
#include <vector>

class AddressBook
{
    public:
    std::vector<string> findMatchingAddresses (std::function<bool (const string&)> func)
    {
        std::vector<string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr )
        {
            // call the function passed into findMatchingAddresses and see if it matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<string> _addresses;
};
```

Lambda function vs. template

- big advantage of std::function over templates is that if you write a template, you need to put the whole function in the header file, whereas std::function does not. This can really help if you're working on code that will change a lot and is included by many source files.

Using std::function

```
std::function<int ()> func;
// check if we have a function (we don't since we didn't provide one)
if ( func )
{
    // if we did have a function, call it
    func();
}
```

Function pointer and lambda functions

```
typedef int (*func)();
func f = [] () -> int { return 2; };
f();
```

Resources

- [1] http://www.cprogramming.com/