LFC Callbacks

Leonard Moşescu
(lemo@metasoft.ro)

1 Introduction

Let's take a look over the **qsort** function, from the C standard library. The (**qsort** function sorts C-style arrays of any type, by making use of the *Quick-Sort* algorithm). Here is what the prototype of this function looks like:

```
void qsort(void *base, size_t nmemb, size_t size,
    int (*compare)(const void *, const void *));
```

As you can see, the **qsort** function has 4 arguments:

base - pointer to the first element of the array to be sorted
nmemb - the number of elements of the array
size - the size of an element
compare - a pointer to a function that takes 2 void pointers as arguments and returns an integer

The last argument is the one that offers great flexibility to the **qsort** function, as the function pointed to by **compare** establishes the order relationship between the array elements, relationship which will be used in the sorting operation. The function pointed to by **compare** takes the addresses of two elements and returns an integer value less than, equal to or greater than 0, if the first element is respectively less than, equal to or greater than the second.

This way, the **qsort** function can be used at sorting integers, strings or objects of any other data type, by allowing the programmers to implement a comparison function for each data type. Moreover, it's possible to write different versions of the comparison function, comparing objects of the same data type by different sorting criteria (e.g. objects of type Employee, can be sorted by their experience, salary, or name).

Let us consider a small example in which we will use the **qsort** function to sort integer arrays:

```
// this function establishes an order relation
int cmpInt(const void *p1, const void *p2)
{
```

```
    return *(const int *)p1 - *(const int *)p2;
}

void main()
{
    int t[100];

    ... here we fill t ...

    // we sort t in ascending order
    qsort(t, 100, sizeof(int), &cmpInt);

    ... here we can use t, as sorted by the cmpInt function ...
}
```

This mechanism, through which the address of a function is passed to another code block, in order to be called by it, is called a **callback**. The **callback** mechanism is a very powerful one, as it enables the parametrisation with the behaviour implemented by an algorithm (module). The author of the **qsort** function, for example, couldn't have known what kind of datatypes would be sorted, or what sorting criteria must be applied at the moment of implemenitng the sorting algorithm, but has allowed for the user of the **qsort** function to write his/her own comparison function,and that function will be called every time two elements need to be sorted (please observe that the functions used as a sorting criteria in **qsort** function can have any name, but they must obey the prototype specified by **qsort**'s designer).

The **callback** mechanism can be used in situations where the user of a module wishes to 'register' his own function, in order to be called from inside the module, for signaling different events or to execute some actions, including the returning of values.

Here are listed some of the most common uses of the **callback** mechanism:

1. **GUI events**
2. **Generic algorithms**
3. **Internal iterators**
4. **System events**

So far we've described the implementation of the **callback** mechanism using pointers to functions in C/C++, an implementation which is very efficient (the time required by a call to a function through a pointer to that function is quite close to a normal function call), but when we are writing code in C++, using classes and objects, we encounter an important limitation: *pointers to functions can only take the address of a global function or that of a static method.* When

using classes and objects in C++, it's useful to be able to use methods of some class as callbacks. Nonstatic methods of a class have a different semantic from the global functions, meaning that a nonstatic method call is associated to an object. In C++, a method call like:

```
object.method(arg1, arg2, ..., argN);
```

has a very close effect to a function call like:

```
function(&object, arg1, arg2, ..., argN);
```

We can see that in the function call we have as an extra argument the address of the object. This extra argument stands as the 'hidden' argument **this** in C++, amd it's present in the call of every nonstatic method. Because of that, pointers to functions can't point to nonstatic methods, even if they do have the same arguments list. C++ doesn't even accept the simulation of a method by explicitly adding **this** argument in the declaration of a pointer to a function. Although we may believe that this trick might work, the semantics of constant objects and of virtual methods would not permit this.

C++ offers us instead pointers to class members. Although their name might fool us, pointers to class members don't represent an absolute address but an offset inside an object. In practice, pointers to members are used together with pointers to objects. Unfortunately, the effective use of a pointer to member - pointer to object pair it's not as easy as the use of simple pointers. The biggest problem is that that in the moment of the declaration of the pointer to member and of the pointer to object, the class of the object must be known. Normally, we'd like to be able to declare the pointers to the methods knowing only the returned type and the arguments list (as soon as some class appears in the pointer's declaration, we won't be able to use anything except objects and methods from that class, and so the flexibility of the **callback** mechanism disappears).

In order to keep the simplicity and flexibility of the **callback** mechanism, in the context of using C++ classes and objects, LFC provides a set of classes that wraps the necessary information needed for global functions calls, of static/nonstatic methods, in a simple and elegant way. The careful use of the powerful features provided by the C++ programming language, especially that of the template mechanism, have lead to the implementation of a kind of **callback** mechanism, which provides some advantages:

- simplicity
- static type checking of the return type and arguments list
- callbacks to functions and to methods have the same semantics
- flexibility (the callbacks can be methods of any C++ class)
- the posibility to provide a suplimentary, constant argument

- the callbacks can return values
- a C++ exception gets thrown when attempting to call a null callback
- calling speed is comparable to that of normal functions/methods
- it's doesn't require language modifications or source processing with external utilities
- natural semantics for callback typed objects: ==, !=, =, copy
- low coupling

2 LFC Callbacks - review

```
////////////////////////////////////////////////////////////////

class IntVector
{
public:
    IntVector() { init(callback(&IntVector::zero)); }

public:
    void init(Callback0<int> cb);
    void forEach(Callback1<void, int> cb);

protected:
    static int zero() { return 0; }

protected:
    int m_data[10];
};

void IntVector::init(Callback0<int> cb)
{
    for(int i = 0; i < 10; i++)
        m_data[i] = cb();
}

void IntVector::forEach(Callback1<void, int> cb)
{
    for(int i = 0; i < 10; i++)
        cb(m_data[i]);
}
```

```
//////////////////////////////////////////////////////////////

class Fibonaci
{
public:
    Fibonaci() { m_a = 1; m_b = 0; }

public:
    int getNext()
    {
        int c = m_a + m_b;
        m_a = m_b;
        m_b = c;
        return m_b;
    }

protected:
    int m_a, m_b;
};

//////////////////////////////////////////////////////////////

class ConsecutiveNumbers
{
public:
    ConsecutiveNumbers() { m_current = 0; }

public:
    int getNext() { return m_current++; }

protected:
    int m_current;
};

//////////////////////////////////////////////////////////////

class Sum
{
public:
    Sum() { m_sum = 0; }

public:
    void addNumber(int number) { m_sum += number; }
    void print() { printf("sum = %d\n", m_sum); }
```

```
protected:
    int m_sum;
};

//////////////////////////////////////////////////////////////////

void printInt(int i)
{
    printf("%d\n", i);
}

//////////////////////////////////////////////////////////////////

int readInt()
{
    int tmp;
    printf(">");
    scanf("%d", &tmp);
    return tmp;
}

//////////////////////////////////////////////////////////////////

void main()
{
    IntVector v;
    Fibonaci fib;
    ConsecutiveNumbers cons;
    Sum sum1, sum2;

    v.init(callback(&readInt));
    v.forEach(callback(&sum1, &Sum::addNumber));
    sum1.print();

    v.init(callback(&cons, &ConsecutiveNumbers::getNext));
    v.forEach(callback(&printInt));
    v.forEach(callback(&sum2, &Sum::addNumber));
    sum2.print();

    v.init(callback(&fib, &Fibonaci::getNext));
    v.forEach(callback(&printInt));
}
```

3 Detailed presentation

3.1 Callback types

LFC offers developers the next 6 template classes categories, for callbacks with cu 0..5 arguments:

```
template<class RT> class Callback0;
template<class RT, class T1> class Callback1;
template<class RT, class T1, class T2> class Callback2;
template<class RT, class T1, class T2, class T3> class Callback3;
template<class RT, class T1, class T2, class T3, class T4> class Callback4;
template<class RT, class T1, class T2, class T3, class T4, class T5> class Callback5;
```

The declaration of an callback object is done as follows:

```
CallbackN<RT, T1, ..., TN> callbackObject;
```

where N represents the number of arguments (0..5), RT is the returned type and T1, ..., TN are the arguments' types. For example, a callback with 2 arguments (an **int** and a **float**) which returns a **bool** will be declared as follows:

```
Callback2<bool, int, float> cb1;
```

A callback without any arguments, which returns a **char ***, will be declared in the following way:

```
Callback0<char *> cb2;
```

You may notice that the callback classes (objects) are used in exactly the same manner as any other C++ template classes (Note: be careful not to use the name **callback** for identifiers, because you may get a conflict with the template function **callback()**).

LFC callback classes were designed in such a manner, so that callback objects may have a natural usage: callback objects can be staticaly, automaticaly (on the free store) or dynamically allocated and can be passed by value or by reference (the copy constructor and operator=() were redefined to ensure the consistency of pass-by-value semantics).

3.2 Calling callback objects

Callback objects have the function call operator defined, in order to allow their use in a simple and elegant way (their calling is done in the same way as calling a function). For example, a callback with an double argument, which returns a long, is called as following:

```
Callback1<long, double> myCallback;

... myCallback initialization with a method/function ...

// callback call
long retValue = myCallback(3.14);
```

3.3 Constructors

Each callback class has more than one template constructor, in order to allow the initialization of a callback object with:

- null callback
- callback to static function/method
- callback to static function/method + aux constant
- callback to non-static method
- callback to non-static method + aux constant
- copy constructor (from a callback of the same type)
- callback to callback + aux constant

3.4 The callback() function

Even if you can use the above constructors for creating callback objects, in practice they are hard to use, because one must completely specify the template class (returned type, arguments' types). That's why LFC provides a simpler alternative to the direct use of constructors: a set of template functions called '**callback()**', which can be used to build callback objects compatible with the function/method passed as a argument. For example:

```
int f(double, double, void *)
{
...
}

void test()
{
    Callback3<int, double, double, void *> cb1;

    // direct use of constructor
    cb1 = Callback3<int, double, double, void *>(&f);
```

```
    // the same effect gained by using the callback() function
    cb1 = callback(&f);

    // in this case, the use of the constructor is preffered
    Callback3<int, double, double, void *> cb2(&f);
    Callback3<int, double, double, void *> cb3 = callback(&f);
}
```

3.5 Callback objects comparison

Two callback objects are considered equal if they are of the same type and if their calling has the same effect (that is if they point to the same function/method and with the same auxiliary value, if any). Testing if two callback objects of the same type are equal/different can be achieved using the operators ==, != (comparing different types of callbacks makes no sense).

Note: In the case of callbacks with an auxiliary value, the comparison can be done only if the operators ==, != are defined for the type of the auxiliary value.

3.6 Null callbacks

The null callbacks are the callback objects that do not point to any function/method. Null callbacks are build by the constructor with empty arguments list.

Testing if a callback is null can be done in the following way:

```
Callback1<void, int> cb;
...
if(cb == Callback1<void, int>())
{
...
}
```

The call of a null callback will generate an exception.

4 LFC Callbacks Reference

```
template<TEMPL> class CallbackN

public:
    CallbackN()

    CallbackN(RT (*pf)(TYPES))

    template<class TAUX>
```

```
    CallbackN(RT (*pf)(TYPES, TAUX), TAUX aux)

    template<class OT, class MT>
    CallbackN(OT *po, RT (MT::*om)(TYPES))

    template<class OT, class MT, class TAUX>
    CallbackN(OT *po, RT (MT::*om)(TYPES, TAUX), TAUX aux)

    CallbackN(const CallbackN<TEMPL> &cb)

    template<class TAUX>
    CallbackN(const CallbackN2<TEMPL, TAUX> &cb, TAUX aux)

    ~CallbackN()

public:
    virtual RT operator()(DECL) const
    const CallbackN<TEMPL> &operator=(const CallbackN<TEMPL> &cb)
    virtual bool operator==(const CallbackN<TEMPL> &cb) const
    virtual bool operator!=(const CallbackN<TEMPL> &cb) const


/*////////////////////////////////////////////////////////////////////*/

template<TEMPL>
CallbackN<TEMPL> callback(RT (*pf)(TYPES))

template<TEMPL, class TAUX>
CallbackN<TEMPL> callback(RT (*pf)(TYPES, TAUX), TAUX aux)

template<class OT, class MT, TEMPL>
CallbackN<TEMPL> callback(OT *po, RT (MT::*om)(TYPES))

template<class OT, class MT, TEMPL, class TAUX>
CallbackN<TEMPL> callback(OT *po, RT (MT::*om)(TYPES, TAUX), TAUX aux)

template<TEMPL>
CallbackN<TEMPL> &callback(const CallbackN<TEMPL> &cb)

template<TEMPL, class TAUX>
CallbackN<TEMPL> callback(const CallbackN2<TEMPL, TAUX> &cb, TAUX aux)
```