Chapter 4
C++ Function Objects or Functors

Lecture Advanced C++
15.04.2014

Pierre Fierz
Bernese University of Applied Sciences
School of Engineering and Information Technology
Division of Computer Science

Contents

1 Introduction
2 How to Replace a Switch-Statement
3 Callbacks
4 Functors
5 Pointers to Class Members

Introduction

• Function Pointers and Function Objects (Functors) are used quite often in C++ programming
• Allows very elegant programming
• Use for
  • Switch / If-Then statement replacement
  • User-defined Late binding
  • Callbacks
• This section copes with
  • Pointers to Functions
  • Function objects or Functor
  • Pointers to Class Members
Points to Functions are used quite extensively in C and C++ programming

- Unfortunately, the syntax is not so easy
- The arguments and the returning type must exactly match between the pointer to the function and its function
- The pointer dereferencation, during the function invocation, is done automatically
- Pointers to functions are not allowed to default arguments!
- There is no generic pointer to function type corresponding to the generic pointer to data type void*.

### Declaration (2)

- Some examples of function pointer declaration

```cpp
int (*fp1)(int);
double *(*fp2)(double *, int);
int (*fp3)(int, char);
double (* arithop[2])(double, double);
```

A function pointer may be initialized or assigned a function address of the same type.

```cpp
double plus(double a, double b) {return a+b;}
double minus(double a, double b) {return a-b;}
double *absvalue(double *a, int c) {
    for(int i = 0; i < c; i++)
        a[i] = std::fabs(a[i]);
    return a;
}

arithop[0] = &plus;
arithop[1] = &plus;
fp2 = &absvalue;
fp1 = &plus; //illegal assignment (type mismatch)
```

### Declaration

- A function pointer is nothing else than a variable, it must be declared as usual.
- The difference to other variables is that you have to declare the whole signature of the function.
- The next statement declares a function which returns an integer and has an integer as parameter

```cpp
int f(int);
```

- To make a pointer out of it, we write a "*" in front of the variable.

```cpp
int *f(int);
```

- Because of operator precedence this is not what we want.
- This is a function which returns a pointer to an integer.
- To get a pointer to the function, we have to use brackets.

```cpp
int (*f)(int);
```

- Now f is a pointer to a function of type int f(int);

### Declaration (2)

- Some examples of function pointer declaration

```cpp
double x[20];
std::cout << arithop[0](16.38, 19.12) << std::endl;
std::cout << arithop[0](16.38, 19.12) << std::endl;
fp2(x);
```
Function pointer as Return Value

- A function pointer can be a function's return value.
- There are two solutions of how to return a pointer to a function.

**Example**

In the next example the function “GetPtr” takes a char argument and returns a pointer to a function which takes to double arguments and returns a double.

```cpp
double (*GetPtr1(const char opCode))(double, double) {
    if (opCode == '+')
        return +;
    else
        return -; // default if invalid operator was passed
}
```

Function pointer as Return Value (2)

- It is more elegant to use a typedef.

**Example**

In the next example we use a typedef to define a pointer to a function which is taking two doubles and returns a double.

```cpp
typedef double(*ptFunc)(double, double);
ptFunc GetPtr2(const char opCode) {
    if (opCode == '+')
        return +;
    else
        return -; // default if invalid operator was passed
}
```

C++ Functors

1 Introduction

2 How to Replace a Switch-Statement

3 Callbacks

4 Functors

5 Pointers to Class Members

Switch-Statement

- In the following example we regard the task to perform one of the four basic arithmetic operation.

**Example**

```cpp
// The four arithmetic operations ... one of these functions
// is selected at runtime with a switch or a function pointer
float Plus(float a, float b) { return a+b; }
float Minus(float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide(float a, float b) { return a/b; }

// The next function use a swith to select the operation
float Switch(float a, float b, char opCode) {
    float result;
    // execute operation
    switch (opCode) {
        case '+': result = Plus (a, b); break;
        case '-': result = Minus (a, b); break;
        case '*': result = Multiply (a, b); break;
        case '/': result = Divide (a, b); break;
    }
    return result;
}
```
Switch-Statement

- In the next function the switch is replaced by a function pointer
- The function pointer "specifies" which operation shall be executed.

Example

```cpp
float With_Function_Pointer(float a, float b, float (*pt2Func)(float, float)) {
    float result = pt2Func(a, b); // call using function pointer
    return result;
}
```

Callback function

- Callback functions (aka Callbacks) are functions that are invoked by a third party (e.g. operating system framework)
- Hollywood Principle: *Don't call us, we'll call you.*
  - E.g. callback that is used when a button is pressed
  - The framework knows when the button is pressed, but it doesn't know what to do at this moment
  - The client (the programmer) customization has to provide the what (to do...)

Callback function (2)

- Consider the following class Button

Example

```cpp
typedef void (*Action)( void ); // callback pointer
class Button {
public:
    Button(const char *label): _label(label), _action(0) {};
    ~Button();
    void press( void ) const {if(_action) _action();}
    void setAction( Action ) { _action = action; }
private:
    string _label;
    Action _action;
};
```
**Callback function (3)**

- The user can customize the behavior of the button

**Customizing behavior**

Button doit("Come on, press me!");
extern void abort(void);
doit.setAction(abort);

- The action can be reset by the client during execution, effectively changing the behavior of the button.

**Change the behavior**

extern void crashSystem(void);
doit.setAction(crashSystem);

- The framework code doesn’t know the action to be performed, but it knows when to execute the action

**Fire the action**

if (lessonIsOver())
doit.press();

---

**Functor Definition**

- A Functor is an object that behaves like a normal function

  - It is:
    - A class...
    - … that defines an operator operator()

- Functors are commonly used for:
  - Encapsulate data (state) with a function
  - Safer and more flexible than normal callback
  - Simplification of the use of pointers to member functions
  - Allow polymorphic behavior
  - Used in STL

---

**C++ Functors**

1. **Introduction**

2. **How to Replace a Switch-Statement**

3. **Callbacks**

4. **Functors**

5. **Pointers to Class Members**

---

**Functor and Callbacks**

- Functor can be used to pass information to a callback function

**Example**

```cpp
// Abstract Action class
class Action {
public:
    virtual ~Action();
    virtual void operator(){};
};
```

- Now the Button class

**Example**

```cpp
class Button {
public:
    Button(string label): _label(label), _action(0) {} 
    virtual ~Button() {}
    void press() const {if (_action) (*_action)(); } 
    void setAction( Action & action) { _action = &action; }
private:
    string _label;
    Action _action;
};
```
Functor and Callbacks (2)

Caution

- The Action object must have a longer lifetime than the button object
- To avoid problems we can clone the Action object

Example

```cpp
class SendFlowers : public Action{
public:
    SendFlowers(string location): _location(location) {}
    void operator()() { cout << "Send flowers to: " << _location << endl; }
private:
    string _location;
    SendFlowers sendMomFlowers("momsHouse"); // Functor
    Button doit("Press me"); //Button
    doit.setAction(sendMomFlowers); // the Functor contains the address and the action!
    // Now send the flowers
    doit.press();
}
```

Now let's apply the new concept

Functors and Algorithms

- Function Pointers and Functors can be used to generate and use Generic Algorithms
- Let's assume a generic algorithm

Example

```cpp
template <class Iter, class Comp>
void slowsort(Iter start, Iter end, Comp less)
{
    for(Iter i(start); i != end; ++i)
        for(Iter j(i); j != end; ++j)
            if(less(*j, *i))
                std::iter_swap(i, j);
}
```

Functors and Algorithms (2)

- The comparison functions (with Function pointers) for strings could look like this:

Example

```cpp
inline bool strLess(const char *a, const char *b)
{
    return strcmp(a, b) < 0;
}
inline bool strLen(const char *a, const char *b)
{
    return strlen(a) < strlen(b);
}
```

So then, the usage would be

Example

```cpp
char *names[100];
//...
slowsort(names, names + 100, strLess);
slowsort(names, names + 100, strLen);
```

Functors and Algorithms (3)

- The comparison function (with Functors) for strings could look like this:

Example

```cpp
class StrLess
{
public:
    bool operator()(const char *a, const char *b)
    {
        return strcmp(a, b) < 0;
    }
};
class StrLen
{
public:
    bool operator()(const char *a, const char *b)
    {
        return strlen(a) < strlen(b);
    }
};
```

So then, the usage would be

Example

```cpp
char *names[100];
//...
slowsort(names, names + 100, StrLess());
slowsort(names, names + 100, StrLen());
```
Functors and Algorithms (4)

- A functor is a full C++ class.
- We can use this fact to solve more complex tasks

Example

```cpp
template <class T> class Sum {
    T result;
public:
    Sum(T i = 0) : result(i) {}  
    void operator() (T x) { result += x; }  
    T getResult() const { return result; }
};
```

- Now we can use this functor to get the sum of a range of iterators.

Example

```cpp
void f(list<double>& ld) {
    Sum<double> s;
    for_each(ld.begin(), ld.end(), s);
    cout << "the sum is: " << s.getResult() << endl;
}
```

Pointers to Class Members

- A Pointer points to a memory location
  - It contains an address
  - The dereferencing of the pointer allows us to access the memory value at this address
  - The dereferencing of the pointer allows us to access the function at this address

Classic Pointers

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Object</th>
</tr>
</thead>
</table>

Member Pointers

```
Object 1
  +----------------+
  | Member         |
  +----------------+
Object 2
  +----------------+
  | Member         |
  +----------------+
```

Pointers to Class Members (2)

- A Pointer to a member is not an address!
  - It does not refer to any particular object, but to its class
  - A pointer to a member is a pointer to a specific member of an unspecified object
  - An object must be supplied to dereference the pointer to the member
    - The pointer can be seen as an offset into a class
    - The offset doesn’t represent a valid member address until it is added to the starting address of a class object
Pointers to Class Members (3)

- Let's assume that we have a class B

**Example**

```cpp
class B
{
    public:
        int i, j;
};
```

- A pointer to member modifier has the form `B::*`

- The declaration syntax is the same as that of any other pointer modifier:

**Example**

```cpp
B cobj;
int *pi = &cobj.i; // Pointer to member of an object
int C::*pimC = &C::i; // Pointer to member of any object
// of type B
```

Pointers to Class Members (4)

- A pointer to a member requires an object to be dereferenced

- The following operators exist:
  - `*` is used with class objects or references
    ```cpp
    int B::pimB = &B::i;
    B cobj;
    int x = cobj.*pimB;
    ```
  - `->*` is used with class pointers
    ```cpp
    B* cp = new B();
    int x = cp->*pimB;
    ```

- Note that `*` and `->*` are single tokens and not two-token sequences

- The operator `*` can not be overloaded (like the single dot operator `.`), whereas `->*` can!

Pointers to Class Members (5)

- There is a predefined conversion from a pointer to a base member to a pointer to a derived member.

- The converse is not true.

- An offset within a derived class may not lie within the base class (e.g D::k)

**Example**

```cpp
class B { int i; int j; };
class D : public B { int k; };
int D::*dptr = &B::i; // OK
int B::*dptr = dptr; // error!
void f( int D::* );
void f( B::* );
f( dptr ); // OK
```

Pointers to Class Members (6)

- Pointers to non-static member functions are declared analogously as pointers to non-member functions

**Example**

```cpp
class B{
    public:
        int i, j;
        void f(void){};
        char* g(int) const {};
    };
B cobj;
B *cp;

void (B::*mfp1)(void) = &B::f;
void (B::* const mfp2)(void) = &B::f;
char* (B::* mfp3)(int) const = &B::g;

(cobj.*mfp1)(); // use it with a specific object
(cp->*mfp1)(); // use it with pointer to an object
```
Pointers to Class Members (7)

- Some side notes about non-static member functions
  - It's impossible to take or call the address of a constructor
  - It's possible to call a destructor explicitly (not really recommended)
  - It's not possible to take the address of the destructor
- Pointers to static members have slightly different rules than pointers to non-static members

Pointers to Static Class Members (2)

- A static member function has no this pointer
- Makes no sense to dereference a static member function in the context of an object

Example

```cpp
class Item{
public:
    static Item* lookup( Key );
    virtual Item* clone() const;
protected:
    void* operator new( size_t );
    void operator delete( void* );
private:
    static Item* _head;
    Key _id;};
```

```
Item* theHead = &Item::_head;
Item* Item::*aHead = &Item::_head; // error
Key* theKey = &Item::_id; // error
Key Item::*aKey = &Item::_id;
```

Pointers to Static Class Members

- A pointer to a static class member is a regular pointer (address)
- A static class member is shared by all objects
- There is a unique address to it (a single object)
- Makes no sense to dereference a static class member in the context of an object

Example

```cpp
class Item{
public:
    static Item* lookup( Key );
    virtual Item* clone() const;
protected:
    void* operator new( size_t );
    void operator delete( void* );
private:
    static Item* _head;
    Key _id;}
```

```
Item* theHead = &Item::_head;
Item* Item::*aHead = &Item::_head; // error
Key* theKey = &Item::_id; // error
Key Item::*aKey = &Item::_id;
```

Pointers to virtual member functions

- Pointers to Virtual Class Members can be seen like pointers to non-static class members
- This is also true for pure virtual functions

Example

```cpp
class Vehicle{
public:
    virtual ~Vehicle();
    virtual void speedup( void ) = 0;
    virtual void slowdown( void ) = 0;
};
```

```
void (Vehicle::*action)(void) = &Vehicle::speedup;
```

```
void (Item::*pmf)(void*) = &Item::operator delete; // error!
void (*pmf2)(void*) = &Item::operator delete; // okay
void* (*fp2)(size_t) = &Item::operator new;
```
The function call depends on the actual type of the class object used to dereference the pointer.

Example

```cpp
class LandVehicle : public Vehicle{
public:
    LandVehicle();
    ~LandVehicle();
    virtual void speedup( void ); // pure in base
    virtual void slowdown( void ); // pure in base
};

LandVehicle vehicle;
Vehicle *tp = &vehicle;
(vehicle.*action)(); // LandVehicle::speedup
(tp->*action)(); // same function
```

Note the notation for pointers to virtual member functions.

Example

```cpp
Vehicle* tp = new LandVehicle();
void (Vehicle::*mfp)(void) = &Vehicle::speedup;
```

Depending on the object pointer the correct function is called

<table>
<thead>
<tr>
<th>Pointer to virtual member</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tp-&gt;*mfp)();</td>
</tr>
</tbody>
</table>

- References to a base class
- References to a derived class

(Inside a base class it is not possible to reference a member defined in a derived class.)