Introduction to C/C++ Function-Pointers, Callbacks and Functors

written by Lars Haendel

www.function-pointer.org

April 2001, Germany

email: lore@newty.de

version 1.04b

Contents

1 Introduction

Function-Pointers provide some extremely interesting, efficient and elegant programming techniques. You can use them to replace *switch/if*-statements, to realize your own *late-binding* or to implement *callbacks*. Unfortunately – probably due to their complicated syntax – they are treated quite stepmotherly in most computer books and documentations. If at all, they are addressed quite briefly and superficially. They are less error prone than normal pointers cause you will never allocate or de-allocate memory with them. All you've got to do is to understand what they are and to learn their syntax. But keep in mind: Always ask yourself if you really need a function-pointer. It's nice to realize one's own late-binding but to use the existing structures of C++ may make your code more readable and clear. One aspect in the case of late-binding is

runtime: If you call a virtual function, your program has got to determine which one has got to be called. It does this using a V-Table containing all the possible functions. This costs some time each call and maybe you can save some time using function-pointers instead of virtual functions. Maybe not ... [1]

1.1 What is a Function-Pointer ?

Function-Pointers are pointers, i.e. variables, which point to the address of a function. You must know, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, have got to be put inside this memory. Thus a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to.

1.2 Why to use them ?

When you want to call a function *DoIt()* at a certain point called *label* in your program, you just put the call of the function *DoIt()* at the point *label* in your source code. Then you compile your code and every time your program comes up to the point *label*, your function is called. Everything is ok. But what can you do, if you don't know at build-time which function has got to be called? What do you do, when you want to decide it at runtime? Maybe you want to use a so called Callback-Function or you want to select one function out of a pool of possible functions dynamically. However you can also solve the latter problem using a *switch*-statement, where you call the functions just like you want it, in the different branches. But there's still another way: Use a function-pointer! In the following example we regard the task to perform one of the four basic arithmetic operations. The task is first solved using a *switch*-statement. Then it is shown, how the same can be done using a function-pointer.[2]

```
// function definitions - one of them is selected at runtime
float Plus    (const float arg1, const float arg2) {return arg1+arg2;}
float Minus   (const float arg1, const float arg2) {return arg1-arg2;}
float Multiply(const float arg1, const float arg2) {return arg1*arg2;}
float Divide  (const float arg1, const float arg2) {return arg1/arg2;}
float arg1=2, arg2=5.5;    // the two arguments for the operation

// task solved using a switch-statement
int main1(){
   char op = '+';          // operation-code

   // calling the correct function which performs the operation
   switch{op}{
      case '+'   : return Plus    (arg1, arg2);      break;  // you won't need the break ;-)
      case '-'   : return Minus   (arg1, arg2);      break;
      case '*'   : return Multiply(arg1, arg2);      break;
      case '/'   : return Divide  (arg1, arg2);      break;
   }
}
```

[1] Modern compilers are very good! With my Borland Compiler the time I was able to save calling a virtual function which multiplies two floats was about 2 percent.

[2] It's only an example and the task is so easy that I suppose nobody will ever use a function-pointer for it ;-)

```
// task solved using as function-pointer
int main2()
{
   // initializing the function-pointer 'opFunc' with the address of 'Plus()'
   // 'opFunc' points to a function which takes two floats and returns an int
   float (*opFunc)(const float, const float) = Plus;

   return opFunc(arg1, arg2);   // call of 'Plus()' using the function-pointer
}
```

Important note: A function-pointer always points to a function with a specific signature! Thus all functions, you want to use with the same function-pointer, must have the **same parameters and return-type!**

2 The Syntax of Function-Pointers

Regarding their syntax, there are two different types of function-pointers: On the one hand there are pointers to ordinary C functions or static C++ member-functions, on the other hand there are pointers to **non-static** C++ member-functions. The basic difference is that all pointers to non-static member functions need a **hidden argument**: The this-pointer to an instance of the class.

2.1 Define a Function-Pointer

Since a function-pointer is nothing else than a variable, it must be defined as usual. In the following example we define a function-pointer named *paraFunc*. It points to a function, which takes one *float* and two *bools* and returns an *int*. In the C++ example it is assumed, that the function, our pointer points to, is a member-function of *TMyClass*.

```
int (*paraFunc)(float, bool, bool);             // C
int (TMyClass::*paraFunc)(float, bool, bool);   // C++
```

2.2 Calling Convention

Normally you don't have to think about a function's calling convention: The compiler assumes *__cdecl* as default if you don't specify another convention. However if you want to know more, keep on reading ... The calling convention tells the compiler things like how to pass the arguments or how to generate the name of a function. Examples for other calling conventions are *__stdcall*, *__pascal*, *__fastcall*. The calling convention belongs to a functions signature: **Thus functions and function-pointers with different calling convention are incompatible with each other!**. You specify a specific calling convention in front of a function's or function-pointer's name but after its return type.[3]

2.3 Assign an Address to a Function-Pointer

It's quite easy to assign the address of a function to a function-pointer. In C you simply take the name of a suitable and known function, in C++ you receive the address of a member-function using the address operator &. Note: You may have got to use the complete name of the member-function including class-name

[3]This is correct for my Borland compiler. I found different explanations on the web and thus I don't know, what's really correct. If someone knows: Let me know ;-)

and scope-operator (::). Also you have got to ensure, that you are allowed to access the function right in scope where your assignment stands.

```
// C
// definition of function 'DoIt'
int DoIt(float arg1, bool arg2, bool arg3){/* do something and return an int */}
paraFunc = DoIt;



// C++
// definition of class TMyClass
class TMyClass{
   int DoIt(float, bool, bool){/* do something and return an int */};
   /* more of TMyClass */
};
paraFunc = &TMyClass::DoIt;
```

Note: You can use the comparison-operator (==) the same way. In the following example it is checked, whether *paraFunc* actually contains the address of the function *DoIt*. A text is shown in case of equality.

```
if(paraFunc == DoIt)               cout << "pointer points to DoIt";            // C
if(paraFunc == &TMyClass::DoIt)    cout << "pointer points to TMyClass::DoIt";  // C++
```

2.4 Calling a Function using a Function-Pointer

In C you call a function using a function-pointer almost the same way, you make a normal function-call. You've just got to use the name of the function-pointer instead of the name of the function. In C++ it's a little bit tricky since you need to have an instance of a class to call one of their (non-static) member-functions. In the following example we assume, that the call takes place within (another) member-function of the class *TMyClass* and thus the *this*-pointer can be used.

```
int para = paraFunc         (12, true, false);    // C
int para = (*this.*paraFunc)(12, true, false);    // C++
```

Now let's call the member via the pointer **from outside** the class. The following code illustrates how to do it. It's important to make the function-pointer *paraFunc* public! Compare the call from outside to the one above. *instance* takes the place of the *this*-pointer and is used to access the function-pointer.

```
// C++
// definition of TMyClass
class TMyClass {
public:
   int (TMyClass::*paraFunc) (float, bool, bool);    // define fpt
   int DoIt(float, bool, bool) { /* do something and return an int */};

   TMyClass() { paraFunc=&TMyClass::DoIt; }          // initialize fpt in constructor
};
```

```
// usage of TMyClass
int test()
{
   TMyClass* instance = new TMyClass;                // new instance

   // call member from outside using the function-pointer
   return (*instance.*instance->paraFunc) (4.2, true, false);

   delete instance;
}
```

2.5 Quite easy: Using Arrays of Function-Pointers

Operating with arrays of function-pointer is very interesting. This offers the possibility to select a function using an index. The syntax appears difficult, which frequently leads to confusion.

```
// C
// type-definition: 'pt2ParaFuncs' now can be used as type, e.g. in a 'new'-statement
typedef int (*pt2ParaFuncs)(float, bool, bool);

// 'funcArray' is a field with pointers to functions with signature: int(float, bool, bool)
pt2ParaFuncs* funcArray;

// create a field with 10 pointers
funcArr = new pt2ParaFunc[10];

// assign the function's address - 'DoIt' and 'DoMore' are suitable functions
funcArr[0] = DoIt;
funcArr[1] = DoMore;
/* more assignments  */

// calling a function using an index to address the function-pointer
int para = funcArr[1](12, true, false);
```

```
// C++
// type-definition: 'pt2ParaFuncs' now can be used as type, e.g. in a 'new'-statement
typedef int (TMyClass::*pt2ParaFuncs)(float, bool, bool);

// 'funcArray' is a field with pointers to member-functions with signature: int(float, bool, bool)
pt2ParaFuncs* funcArray;

// create a field with 10 pointers
funcArr = new pt2ParaFunc[10];
```

```
// assign the function's address - 'DoIt' and 'DoMore' are suitable member-functions
// of the class TMyClass
funcArr[0] = &TMyClass::DoIt;
funcArr[1] = &TMyClass::DoMore;
/* more assignments */

// calling a function using an index to address the function-pointer
int para = (*this.*funcArr[1])(12, true, false);
```

3 Callback Functions

3.1 Introduction to the Concept of Callback Functions

Function-Pointers provide the concept of callback functions. I'll try to introduce the concept of callback functions using the well known sort function *qsort*. This function sorts the items of a field according to a user-specific ranking. The field can contain items of any type; it is passed to the sort function using a *void*-pointer. Also the size of an item and the total number of items in the field has got to be passed. Now the question is: How can the sort-function sort the items of the field without any information about the type of an item? The answer is simple: The function receives the pointer to a comparison-function which takes *void*-pointers of two field-items, evaluates their ranking and returns the result coded as an *int*. So every time the sort algorithm needs a decision about the ranking of two items, it just calls the comparison-function via the function-pointer.

3.2 How to Implement a Callback in C ?

To explain I just take the declaration of the function *qsort* which reads itself as follows[4]:

```
void qsort(void* field, size_t nElements, size_t sizeOfAnElement,
           int(_USERENTRY *cmpFunc)(const void *, const void*));
```

field points to the first element of the field which is to be sorted, *nElements* is the number of items in the field, *sizeOfAnElement* the size of one item in bytes and *cmpFunc* is the pointer to the comparison function. This comparison function takes two *void*-pointers and returns an *int*. The syntax, how you use a function-pointer as a parameter in a function-definition looks a little bit strange. Just review, how to define a function-pointer and you'll see, it's exactly the same. A **callback is done** just like a normal function call would be done: You just use the name of the function-pointer instead of a function name. This is shown below. Note: All calling arguments despite the function-pointer were omitted to focus on the relevant things.

```
void qsort( ... , int(_USERENTRY *cmpFunc)(const void*, const void*))
{
   /* sort algorithm  - note: item1 and item2 are void-pointers */

   bool bigger=cmpFunc(item1, item2);  // make callback

   /* use the result */
}
```

[4]Taken from the Compiler Borland C++ 5.02 (BC5.02)

3.3 Example Code of the Usage of *qsort*

```
// header files if you use the compiler BC5.02
#include <stdlib.h>     // for qsort()
#include <time.h>       // for randomize()

// comparison-function for the sort-algorithm
// two items are taken by void-pointer, converted and compared
int cmpFunc(const void* _a, const void* _b)
{
   // you've got to explicitly cast to the correct type
   const float* a = (const float*) _a;
   const float* b = (const float*) _b;

   if(*a > *b)  return 1;       // first item is bigger than the second one -> return 1
   else
      if(*a == *b)  return  0;  // equality -> return 0
      else          return -1;  // second item is bigger than the first one -> return -1
}

// example for the use of qsort()
void example()
{
   float* field=new float[1000];

   ::randomize();              // initialize random-number-generator
   for(int c=0;c<1000;c++)     // randomize all items of the field
      field[c]=random(99);

   // sort using qsort()
   qsort((void*) field, /*number of items*/ 1000, /*size of an item*/ sizeof(field[0]),
           /*comparison-function*/ cmpFunc);

    /* do something useless ;-) with 'field' */

   delete[] field;
}
```

3.4 How to Implement a Callback to a static C++ Member-Function ?

This is the same as you implement callbacks to C functions. Static member-functions do not need an object to be invoked on and thus have the same signature as a C function with the same calling convention, calling arguments and return type.

3.5 How to Implement a Callback to a non-static C++ Member-Function ?

Pointers to non-static members are different to ordinary C function-pointers since they need the this-pointer of a class object to be passed. Thus ordinary function-pointers and non-static member-functions have different and incompatible signatures! If you just want to callback to a member of a specific class you just change your code from an ordinary function-pointer to a pointer to a member-function. But what can you do, if you want to **callback to a non-static member of an arbitrary class**? It's a little bit difficult. You need to write a **static** member-function as a wrapper. A static member-function has the same signature as a C function! Then you cast the pointer to the object on which you want to invoke the member-function to **void*** and pass it to the wrapper as an **additional argument** or via a **global variable**.[5] Of course you've also got to pass the calling arguments for the member-function. The wrapper casts the void-pointer to a pointer to an instance of the correct class and calls the member-function.

Example: Pointer to a class instance passed as an additional argument The function *DoIt* does something with objects of the class *TClassA* which implies a callback. Therefore a pointer to an object of class *TClassA* and a pointer to the static wrapper function *TClassA::Wrapper_To_Call_Display* are passed to *DoIt*. This wrapper is the callback-function. You can write arbitrary other classes like *TClassA* and use them with *DoIt* as long as these other classes provide the necessary functions.

```
//  we want to callback to the member Display(), therefore a wrapper is used
class TClassA
{
public:
   void Display(const char* text) { /* display something */ };
   static void Wrapper_To_Call_Display(void* pt2Object, char* text);

   /* more of TClassA */
};

// wrapper to be able to callback the member Display()
void TClassA::Wrapper_To_Call_Display(void* pt2Object, char* string)
{
   // explicitly cast to a pointer to TClassA
   TClassA* mySelf = (TClassA*) pt2Object;

   // call member
   mySelf->Display(string);
}

// function does something which implies a callback
// note: of course this function can also be a member-function
void DoIt(void* pt2Object, void (*callback)(void* pt2Object, char* text))
{
   /* do something */
   callback(pt2Object, "hi, i'm calling back ;-)");      // callback
}
```

[5] If you use a global variable it is very important that you make sure that it will always point to the correct objects!

```
// main program
void test()
{
   // 1. instantiate object of TClassA
   TClassA objA;

   // 2. call DoIt for <objA>
   DoIt((void*) &objA, TClassA::Wrapper_To_Call_Display);
}
```

Example: Pointer to a class instance is stored in a global variable The function *DoIt* does something with objects of the class *TClassA* which implies a callback. A pointer to the static wrapper function *TClassA::Wrapper_To_Call_Display* is passed to *DoIt*. This wrapper is the callback-function. The wrapper uses the global variable *void* pt2Object* and explicitly casts it to an instance of *TClassA*. It is very important, that you always initialize the global variable to point to the correct class instance. You can write arbitrary other classes like *TClassA* and use them with *DoIt* as long as these other classes provide the necessary functions.

```
void* pt2Object;  // global variable which points to an arbitrary object

//  we want to callback to the member Display(), therefore a wrapper is used
class TClassA
{
public:
   void Display(const char* text) { /* display something */ };
   static void Wrapper_To_Call_Display(char* text);

   /* more of TClassA */
};

// wrapper to be able to callback the member Display()
void TClassA::Wrapper_To_Call_Display(char* string)
{
    // explicitly cast global variable <pt2Object> to a pointer to TClassA
    // warning: <pt2Object> must point to an appropriate object!
    TClassA* mySelf = (TClassA*) pt2Object;

    // call member
    mySelf->Display(string);
}

// function does something which implies a callback
// note: of course this function can also be a member-function
void DoIt(void (*callback)(void* pt2Object, char* text))
{
```

```
   /* do something */

   callback("hi, i'm calling back ;-)");      // callback
}

// main program
void test()
{
   // 1. instantiate object of TClassA
   TClassA objA;

   // 2. assign global variable which is used in the static wrapper function
   // important: never forget to do this!!
   pt2Object = (void*) &objA;

   // 3. call DoIt for <objA>
   DoIt(TClassA::Wrapper_To_Call_Display);
}
```

4 Functors to encapsulate Function-Pointers in C and C++

4.1 What are Functors ?

Functors can encapsulate function-pointers in C and C++ using templates and polymorphism. Thus you can build up a list of pointers to member-functions of arbitrary classes and call them all through the same interface without bothering about their class or the need of a pointer to an instance. All the functions just have got to have the same return-type and calling parameters. Sometimes Functors are also known as Closures. You can use Functors to implement callbacks.

4.2 How to Implement Functors ?

First you need a base class **TFunctor** which provides a virtual function named *Call* or a virtually overloaded operator () with which you will be able to call the member-function. It's up to you if you prefer the overloaded operator or a function like *Call*. From the base class you derive a **template class TSpecificFunctor** which is initialized with a pointer to an object and a pointer to a member-function in its constructor. **The derived class overrides the function *Call* and the operator () of the base class:** In the overrided versions it calls the member-function using the stored pointers to the object and to the member-function.

```
// abstract base class
class TFunctor
{
public:
   // two possible functions to call member-function. virtual cause derived
   // classes will use a pointer to an object and a pointer to a member-function
   // to make the function call
   virtual void operator()(const char* string){};  // call using operator
   virtual void Call(const char* string) {};       // call using function
```

```
};

// derived template class
template <class TClass> class TSpecificFunctor : public TFunctor
{
private:
   void (TClass::*fpt)(const char*);   // pointer to member-function
   TClass* pt2Object;                  // pointer to object

public:
   // constructor - takes pointer to an object and pointer to a member and stores
   // them in two private variables
   TSpecificFunctor(TClass* _pt2Object, void(TClass::*_fpt)(const char*))
      { pt2Object = _pt2Object;  fpt=_fpt; };

   // override operator "()"
   virtual void operator()(const char* string)
    { (*pt2Object.*fpt)(string);};              // execute member-function

   // override function "Call"
   virtual void Call(const char* string)
     { (*pt2Object.*fpt)(string);};             // execute member-function
};
```

4.3 Example of How to Use Functors

In the following example we have two dummy classes which provide a function called Display which returns nothing (void) and needs a string (const char*) to be passed. We create an array with two pointers to **TFunctor** and initialize the array entries with two pointers to **TSpecificFunctor** which encapsulate the pointer to an object and the pointer to a member of TClassA respectively TClassB. Then we use the functor-array to call the respective member-functions. **No pointer to an object is needed to make the function calls and you do not have to bother about the classes anymore!**

```
// dummy class A
class TClassA{
public:
   TClassA(){};
   void Display(const char* text) { /* display something somehow */ };

   /* more of TClassA */
};

// dummy class B
class TClassB{
public:
   TClassB(){};
   void Display(const char* text) { /* display something somehow */ };
```

```
   /* more of TClassB */
};

// main program - illustrate how to use the functors
void main1()
{
   // 1. instantiate objects of TClassA and TClassB
   TClassA objA();
   TClassB objB();

   // 2. create array with pointers to TFunctor, the base class
   TFunctor** vTable = new TFunctor*[2];

   // 3. instantiate TSpecificFunctor objects ...
   //    a ) functor which encapsulates pointer to object and to member of TClassA
   TSpecificFunctor<TClassA> specFuncA(&objA, &TClassA::Display);

   //    b) functor which encapsulates pointer to object and to member of TClassB
   TSpecificFunctor<TClassB> specFuncB(&objB, &TClassB::Display);

   // assign their addresses to the function-pointer array
   vTable[0] = &specFuncA;
   vTable[1] = &specFuncB;

   // 4. use array to call member-functions without the need of an object
   vTable[0]->Call("TClassA::Display called!");        // via function "Call"
   (*vTable[1])   ("TClassB::Display called!");        // via operator "()"

   // 5. release
   delete[] vTable;
}
```