The Function Pointer Tutorials

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

written by Lars Haendel
January 2005, Bochum, Germany
http://www.newty.de
email: Have a look at the web page please
version 2.07

Copyright (c) 2000-2005 by Lars Haendel. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Text being the text from the title up to the table of contents, and with no Back-Cover Texts. A copy of the license can be obtained from http://www.gnu.org .

Be aware that there may be a newer version of this document! Check http://www.newty.de/fpt/zip/e_fpt.pdf for the latest release. If you want to distribute this document, I suggest you to link to the URL above to prevent spreading of outdated versions.

You may also download the source code of the examples at http://www.newty.de/fpt/zip/source.zip . The example code is free software under the terms of the GNU General Public License.

Contents

1 Introduction to Function Pointers

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 keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are 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 Introductory Example or How to Replace a *Switch*-Statement

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. 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]

```
//------------------------------------------------------------------------------------
// 1.2 Introductory Example or How to Replace a Switch-Statement
// Task: Perform one of the four basic arithmetic operations specified by the
//       characters '+', '-', '*' or '/'.

// The four arithmetic operations ... one of these functions is selected
// at runtime with a swicth 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; }

// Solution with a switch-statement - <opCode> specifies which operation to execute
void 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; }
```

[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 ;-)

```
   cout << "Switch: 2+5=" << result << endl;         // display result
}

// Solution with a function pointer - <pt2Func> is a function pointer and points to
// a function which takes two floats and returns a float. The function pointer
// "specifies" which operation shall be executed.
void Switch_With_Function_Pointer(float a, float b, float (*pt2Func)(float, float))
{
   float result = pt2Func(a, b);    // call using function pointer
   cout << "Switch replaced by function pointer: 2-5=";  // display result
   cout << result << endl;
}

// Execute example code
void Replace_A_Switch()
{
   cout << endl << "Executing function 'Replace_A_Switch'" << endl;
   Switch(2, 5, /* '+' specifies function 'Plus' to be executed */ '+');
   Switch_With_Function_Pointer(2, 5, /* pointer to function 'Minus' */ &Minus);
}
```

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 C and C++ 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 to 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. Always keep in mind: These two types of function pointers are incompatible with each other.

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 two function pointers named *pt2Function*, *pt2Member* and *pt2ConstMember*. They point to functions, which take one *float* and two *char* and return an *int*. In the C++ example it is assumed, that the functions, our pointers point to, are (non-static) member functions of *TMyClass*.

```
// 2.1 define a function pointer and initialize to NULL
int (*pt2Function)(float, char, char) = NULL;                        // C
int (TMyClass::*pt2Member)(float, char, char) = NULL;                // C++
int (TMyClass::*pt2ConstMember)(float, char, char) const = NULL; // 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!** For Borland and Microsoft compilers you specify a specific calling convention between the return type and the function's or function pointer's name. For the GNU GCC you use the *__attribute__* keyword: Write the function definition followed by the keyword *__attribute__* and then state the calling convention in double parentheses.[3]

[3] If someone knows more: Let me know ;-) And if you want to know how function calls work under the hood you should take a look at the chapter *Subprograms* in Paul Carter's *PC Assembly Tutorial* (http://www.drpaulcarter.com/pcasm/).

```
// 2.2 define the calling convention
void __cdecl DoIt(float a, char b, char c);                         // Borland and Microsoft
void         DoIt(float a, char b, char c) __attribute__((cdecl));  // GNU GCC
```

2.3 Assign an Address to a Function Pointer

It's quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. Although it's optional for most compilers you should use the address operator `&` infront of the function's name in order to write portable code. You may have got to use the complete name of the member function including class-name and scope-operator (`::`). Also you have got to ensure, that you are allowed to access the function right in scope where your assignment stands.

```
// 2.3 assign an address to the function pointer
//     Note: Although you may ommit the address operator on most compilers
//     you should always use the correct way in order to write portable code.

// C
int DoIt  (float a, char b, char c){ printf("DoIt\n");   return a+b+c; }
int DoMore(float a, char b, char c)const{ printf("DoMore\n"); return a-b+c; }

pt2Function = DoIt;      // short form
pt2Function = &DoMore;   // correct assignment using address operator


// C++
class TMyClass
{
public:
   int DoIt(float a, char b, char c){ cout << "TMyClass::DoIt"<< endl; return a+b+c;};
   int DoMore(float a, char b, char c) const
         { cout << "TMyClass::DoMore" << endl; return a-b+c; };

   /* more of TMyClass */
};

pt2ConstMember = &TMyClass::DoMore; // correct assignment using address operator
pt2Member = &TMyClass::DoIt; // note: <pt2Member> may also legally point to &DoMore
```

2.4 Comparing Function Pointers

You can use the comparison-operators (`==`, `!=`) the same way as usual. In the following example it is checked, whether *pt2Function* and *pt2Member* actually contain the address of the functions *DoIt* and *TMyClass::DoMore*. A text is shown in case of equality.

```
// 2.4 comparing function pointers

// C
if(pt2Function >0){                           // check if initialized
   if(pt2Function == &DoIt)
      printf("Pointer points to DoIt\n"); }
else
   printf("Pointer not initialized!!\n");


// C++
if(pt2ConstMember == &TMyClass::DoMore)
   cout << "Pointer points to TMyClass::DoMore" << endl;
```

2.5 Calling a Function using a Function Pointer

In C you call a function using a function pointer by explicitly dereferencing it using the `*` operator. Alternatively you may also just use the function pointer's instead of the funtion's name. In C++ the two operators `.*` resp. `->*` are used together with an instance of a class in order to call one of their (non-static) member functions. If the call takes place within another member function you may use the *this*-pointer.

```
// 2.5 calling a function using a function pointer
int result1 = pt2Function    (12, 'a', 'b');          // C short way
int result2 = (*pt2Function) (12, 'a', 'b');          // C

TMyClass instance1;
int result3 = (instance1.*pt2Member)(12, 'a', 'b');   // C++
int result4 = (*this.*pt2Member)(12, 'a', 'b');       // C++ if this-pointer can be used

TMyClass* instance2 = new TMyClass;
int result4 = (instance2->*pt2Member)(12, 'a', 'b');  // C++, instance2 is a pointer
delete instance2;
```

2.6 How to Pass a Function Pointer as an Argument ?

You can pass a function pointer as a function's calling argument. You need this for example if you want to pass a pointer to a callback function. The following code shows how to pass a pointer to a function which returns an int and takes a float and two char:

```
//-----------------------------------------------------------------------------------------
// 2.6 How to Pass a Function Pointer

// <pt2Func> is a pointer to a function which returns an int and takes a float and two char
void PassPtr(int (*pt2Func)(float, char, char))
{
   int result = (*pt2Func)(12, 'a', 'b');     // call using function pointer
   cout << result << endl;
}

// execute example code - 'DoIt' is a suitable function like defined above in 2.1-4
void Pass_A_Function_Pointer()
{
   cout << endl << "Executing 'Pass_A_Function_Pointer'" << endl;
   PassPtr(&DoIt);
}
```

2.7 How to Return a Function Pointer ?

It's a little bit tricky but a function pointer can be a function's return value. In the following example there are two solutions of how to return a pointer to a function which is taking two *float* arguments and returns a *float*. If you want to return a pointer to a member function you have just got to change the definitions/declarations of all function pointers.

```
//-----------------------------------------------------------------------------------------
// 2.7 How to Return a Function Pointer
//     'Plus' and 'Minus' are defined above. They return a float and take two float

// Direct solution: Function takes a char and returns a pointer to a
// function which is taking two floats and returns a float. <opCode>
// specifies which function to return
float (*GetPtr1(const char opCode))(float, float){
   if(opCode == '+')
      return &Plus;
   else
      return &Minus;} // default if invalid operator was passed
```

```
// Solution using a typedef: Define a pointer to a function which is taking
// two floats and returns a float
typedef float(*pt2Func)(float, float);

// Function takes a char and returns a function pointer which is defined
// with the typedef above. <opCode> specifies which function to return
pt2Func GetPtr2(const char opCode)
{
   if(opCode == '+')
      return &Plus;
   else
      return &Minus; // default if invalid operator was passed
}

// Execute example code
void Return_A_Function_Pointer()
{
   cout << endl << "Executing 'Return_A_Function_Pointer'" << endl;

   // define a function pointer and initialize it to NULL
   float (*pt2Function)(float, float) = NULL;

   pt2Function=GetPtr1('+');   // get function pointer from function 'GetPtr1'
   cout << (*pt2Function)(2, 4) << endl;   // call function using the pointer


   pt2Function=GetPtr2('-');   // get function pointer from function 'GetPtr2'
   cout << (*pt2Function)(2, 4) << endl;   // call function using the pointer
}
```

2.8 How to Use 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. Below you find two ways of how to define and use an array of function pointers in C and C++. The first way uses a *typedef*, the second way directly defines the array. It's up to you which way you prefer.

```
//------------------------------------------------------------------------------------
// 2.8 How to Use Arrays of Function Pointers

// C ---------------------------------------------------------------------------------

// type-definition: 'pt2Function' now can be used as type
typedef int (*pt2Function)(float, char, char);

// illustrate how to work with an array of function pointers
void Array_Of_Function_Pointers()
{
   printf("\nExecuting 'Array_Of_Function_Pointers'\n");

   // define arrays and ini each element to NULL, <funcArr1> and <funcArr2> are arrays
   // with 10 pointers to functions which return an int and take a float and two char

   // first way using the typedef
   pt2Function funcArr1[10] = {NULL};

   // 2nd way directly defining the array
   int (*funcArr2[10])(float, char, char) = {NULL};
```

```
   // assign the function's address - 'DoIt' and 'DoMore' are suitable functions
   // like defined above in 2.1-4
   funcArr1[0] = funcArr2[1] = &DoIt;
   funcArr1[1] = funcArr2[0] = &DoMore;

   /* more assignments */

   // calling a function using an index to address the function pointer
   printf("%d\n", funcArr1[1](12, 'a', 'b'));         //  short form
   printf("%d\n", (*funcArr1[0])(12, 'a', 'b'));      // "correct" way of calling
   printf("%d\n", (*funcArr2[1])(56, 'a', 'b'));
   printf("%d\n", (*funcArr2[0])(34, 'a', 'b'));
}

// C++ -------------------------------------------------------------------------------

// type-definition: 'pt2Member' now can be used as type
typedef int (TMyClass::*pt2Member)(float, char, char);

// illustrate how to work with an array of member function pointers
void Array_Of_Member_Function_Pointers()
{
   cout << endl << "Executing 'Array_Of_Member_Function_Pointers'" << endl;

   // define arrays and ini each element to NULL, <funcArr1> and <funcArr2> are
   // arrays with 10 pointers to member functions which return an int and take
   // a float and two char

   // first way using the typedef
   pt2Member funcArr1[10] = {NULL};

   // 2nd way of directly defining the array
   int (TMyClass::*funcArr2[10])(float, char, char) = {NULL};


   // assign the function's address - 'DoIt' and 'DoMore' are suitable member
   //  functions of class TMyClass like defined above in 2.1-4
   funcArr1[0] = funcArr2[1] = &TMyClass::DoIt;
   funcArr1[1] = funcArr2[0] = &TMyClass::DoMore;
   /* more assignments */

   // calling a function using an index to address the member function pointer
   // note: an instance of TMyClass is needed to call the member functions
   TMyClass instance;
   cout << (instance.*funcArr1[1])(12, 'a', 'b') << endl;
   cout << (instance.*funcArr1[0])(12, 'a', 'b') << endl;
   cout << (instance.*funcArr2[1])(34, 'a', 'b') << endl;
   cout << (instance.*funcArr2[0])(89, 'a', 'b') << endl;
}
```

3 How to Implement Callback Functions in C and C++

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 to 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 other than 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 */

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

   /* use the result */
}
```

3.3 Example Code of the Usage of *qsort*

```
//-----------------------------------------------------------------------------------------
// 3.3 How to make a callback in C by the means of the sort function qsort

#include <stdlib.h>     // due to:  qsort
#include <time.h>       //          randomize
#include <stdio.h>      //          printf

// 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 QSortExample()
{
   float field[100];

   ::randomize();                  // initialize random-number-generator
   for(int c=0;c<100;c++)          // randomize all elements of the field
```

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

```
      field[c]=random(99);

   // sort using qsort()
   qsort((void*) field, /*number of items*/ 100, /*size of an item*/ sizeof(field[0]),
         /*comparison-function*/ CmpFunc);

   // display first ten elements of the sorted field
   printf("The first ten elements of the sorted field are ...\n");
   for(int c=0;c<10;c++)
      printf("element #%d contains %.0f\n", c+1, field[c]);
   printf("\n");
}
```

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. Below you find two examples:

Example A: Pointer to a class instance passed as an additional argument The function *DoItA* 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 *DoItA*. This wrapper is the callback-function. You can write arbitrary other classes like *TClassA* and use them with *DoItA* as long as these other classes provide the necessary functions. **Note:** This solution may be useful if you design the callback interface yourself. It is much better than the second solution which uses a global variable.

```
//-----------------------------------------------------------------------------------------
// 3.5 Example A: Callback to member function using an additional argument
// Task: The function 'DoItA' makes something which implies a callback to
//       the member function 'Display'. Therefore the wrapper function
//       'Wrapper_To_Call_Display is used.

#include <iostream.h>   // due to:   cout

class TClassA
{
public:

   void Display(const char* text) { cout << text << endl; };
   static void Wrapper_To_Call_Display(void* pt2Object, char* text);

   /* more of TClassA */
};
```

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

```
// static wrapper function to be able to callback the member function 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 DoItA(void* pt2Object, void (*pt2Function)(void* pt2Object, char* text))
{
   /* do something */

   pt2Function(pt2Object, "hi, i'm calling back using a argument ;-)");  // make callback
}

// execute example code
void Callback_Using_Argument()
{
   // 1. instantiate object of TClassA
   TClassA objA;

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

Example B: Pointer to a class instance is stored in a global variable The function *DoItB* does something with objects of the class *TClassB* which implies a callback. A pointer to the static wrapper function *TClassB::Wrapper_To_Call_Display* is passed to *DoItB*. This wrapper is the callback-function. The wrapper uses the global variable *void* pt2Object* and explicitly casts it to an instance of *TClassB*. 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 *TClassB* and use them with *DoItB* as long as these other classes provide the necessary functions. **Note:** This solution may be useful if you have an existing callback interface which cannot be changed. It is not a good solution because the use of a global variable is very dangerous and could cause serious errors.

```
//-----------------------------------------------------------------------------------------
// 3.5 Example B: Callback to member function using a global variable
// Task: The function 'DoItB' makes something which implies a callback to
//       the member function 'Display'. Therefore the wrapper function
//       'Wrapper_To_Call_Display is used.

#include <iostream.h>   // due to:   cout

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

class TClassB
{
public:
   void Display(const char* text) { cout << text << endl; };
   static void Wrapper_To_Call_Display(char* text);

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

```
// static wrapper function to be able to callback the member function Display()
void TClassB::Wrapper_To_Call_Display(char* string)
{
    // explicitly cast global variable <pt2Object> to a pointer to TClassB
    // warning: <pt2Object> MUST point to an appropriate object!
    TClassB* mySelf = (TClassB*) 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 DoItB(void (*pt2Function)(char* text))
{
   /* do something */
   pt2Function("hi, i'm calling back using a global ;-)");   // make callback
}

// execute example code
void Callback_Using_Global()
{
   // 1. instantiate object of TClassB
   TClassB objB;

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

   // 3. call 'DoItB' for <objB>
   DoItB(TClassB::Wrapper_To_Call_Display);
}
```

4 Functors to encapsulate C and C++ Function Pointers

4.1 What are Functors ?

Functors are *functions with a state*. In C++ you can realize them as a class with one or more private members to store the state and with an overloaded operator[6] () to execute the function. Functors can encapsulate C and C++ function pointers employing the concepts templates and polymorphism. 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 also 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/or 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.

[6] If you prefer you can also use a function called *Execute* or something like that.

```
//-----------------------------------------------------------------------------------------
// 4.2 How to Implement Functors

// 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)=0;  // call using operator
   virtual void Call(const char* string)=0;        // 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!**

```
//-----------------------------------------------------------------------------------------
// 4.3 Example of How to Use Functors

// dummy class A
class TClassA{
public:
   TClassA(){};
   void Display(const char* text) { cout << text << endl; };
   /* more of TClassA */
};
```

```
// dummy class B
class TClassB{
public:
   TClassB(){};
   void Display(const char* text) { cout << text << endl; };
   /* more of TClassB */
};

// main program
int main(int argc, char* argv[])
{
   // 1. instantiate objects of TClassA and TClassB
   TClassA objA;
   TClassB objB;

   // 2. 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);

   // 3. make array with pointers to TFunctor, the base class, and initialize it
   TFunctor* vTable[] = { &specFuncA, &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 "()"


   cout << endl << "Hit Enter to terminate!" << endl;
   cin.get();
   return 0;
}
```