The Function Pointer Tutorials

The Function Pointer Tutorials

3. How to Implement Callbacks in C and C++

3.1 Introduction to the Concept of Callback Functions

Function Pointers provide the concept of callback functions. If you are not sure of how to use function pointers take a look at the **Introduction to Function Pointers**. 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: It makes a callback!

3.2 How to Implement a Callback in C ?

I just take the declaration of the function *qsort* which reads itself as follows:

```
   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*

Index Top

In the following example a field of *floats* is sorted.

```
//-----------------------------------------------------------------------------------------
// 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
      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 ?

Index Top

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 ?

Index Top

The Wrapper Approach

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**. If you use a global variable it is very important, that you make sure that it will always point to the correct object! 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 corresponding class and calls the member function. Below you find two examples.

Example A: Pointer to a class instance is 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. **Nore:** This solution may be useful if you design the callback interface yourself. It is much better than the second solution which uses as 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 */
};


// 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

Index Top

The function **DoItB** does something with objects of the class TClassB which implies a callback. A pointer to the static 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);
}
```

Index Top

You can download a Postscript or a PDF of the tutorials about Function Pointers, Callbacks and Functors!

Vous pouvez télécharger une version Postscript ou PDF de ce turoriel sur les pointeurs de fonctions, les fonction de rappel et les *functors*!

You can also download the zipped source-files with all examples of the tutorials!