The Function Pointer

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.

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-tatement, 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.

```
//------------------------------------------------------------------------------
// 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 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; } // b!=0 is assumed
```

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

//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.

float Switch_With_Function_Pointer(float a, float b, float (*pt2Func)(float,
float))
{
      float result = pt2Func(a, b); // call using function pointer

      return result;
}

// Execute example code
void Replace_A_Switch()
{
      float result1, result2;
      printf("\nExecuting function 'Replace_A_Switch'\n");

      result1 = Switch(2, 5, /* '+' specifies function 'Plus' to be executed */
      '+');

      result2 = 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 Function Pointers

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 pointers named pt2Function. It points to a function, which take one float and two char and return an int.

```
// 2.1 define a function pointer and initialize to NULL
int (*pt2Function)(float, char, char) = NULL;
```

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.

```
// 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.

```
// 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.

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


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

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 actually contains the address of the function DoIt.
A text is shown in case of equality.

```
// 2.4 comparing function pointers

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

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.

```
// 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
```

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
      printf("%d", result);
}

// execute example code -'DoIt' is a suitable function like defined above in
2.1-4
void Pass_A_Function_Pointer()
{
      printf("Executing 'Pass_A_Function_Pointer'\n");
      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.

```
//------------------------------------------------------------------------------
// 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(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(char opCode)
{
      if(opCode == '+')
            return &Plus;
      else
            return &Minus; // default if invalid operator was passed
}

// Execute example code
void Return_A_Function_Pointer()
{
      printf("Executing 'Return_A_Function_Pointer'\n");

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

      pt2Function=GetPtr1('+'); // get function pointer from function 'GetPtr1'
      printf("%d", (*pt2Function)(2, 4)); // call function using the pointer

      pt2Function=GetPtr2('-'); // get function pointer from function 'GetPtr2'
      printf("%d", (*pt2Function)(2, 4)); // 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. 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
// ------------------------------------------------------------------------
// 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'));
}
```

3 How to Implement Callback Functions in 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:

```
void qsort(void* Arr, int nElements, int sizeOfAnElement,
           int(*cmpFunc)(void *, void*));
```

Arr 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(*cmpFunc)(void*, 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 CmpFuncFloat(void* itemA, void* itemB)
{

      // you've got to explicitly cast to the correct type

      float* a = (float*) itemA;

      float* b = (float*) itemB;

      // first item is bigger than the second one -> return 1
      if(*a > *b)
            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 Arr[100];
      int c;

      ::randomize(); // initialize random-number-generator
      for(int c=0;c<100;c++) // randomize all elements of the field
            Arr[c]=random(99);

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

      // 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, Arr[c]);
      printf("\n");
}
```