Missing Parts

Jan Gaura

10. 12. 2019

Outline

- Function pointers (aka functional C)
- Administrative

We Love Pointes in C :-)

- Pointer addresses memory on heap

```
int * p = NULL;

p = (int*)malloc( N * sizeof( p[ 0 ] ) );

// name `p` is pointing to allocated memory chunk
```

- Pointers can be passed around, e.g., function arguments

Can Other Names Point to Something Else?

- In Haskell, function can be passed as an argument to other functions

Can Other Names Point to Something Else?

- In Haskell, function can be passed as an argument to other functions
- This is one of the fundamental properties of FP

Can Other Names Point to Something Else?

- In Haskell, function can be passed as an argument to other functions
- This is one of the fundamental properties of FP
- Other may be no side effects (quite impossible to do in C/C++)

Well..

- In C, function name is pointer as well

Well..

- In C, function name is pointer as well
- So is C a functional programming language?

Well..

- In C, function name is pointer as well
- So is C a functional programming language?
- Or course it is NOT...

Well..

- In C, function name is pointer as well
- So is C a functional programming language?
- Or course it is NOT…
- …but we can use function pointers to abstract our code

Compute Square of Numbers

- *Problem:* Compute a square of each element in an array of numbers.

Compute Square of Numbers

- *Problem:* Compute a square of each element in an array of numbers.
- *Input:* Array of integers of known length.

Compute Square of Numbers

- *Problem:* Compute a square of each element in an array of numbers.
- *Input:* Array of integers of known length.
- *Ouput:* Array of squared integers at correspongin indices (same length as input array).

Square Function

```
int sqr( const int x ) {
    return x * x;
}
```

Loop Over an Array

```
const int N = 5;
int array_in[ N ];
int array_out [ N ];

// init array
for ( int i = 0; i < N; i++ ) {
    array_in[ i ] = i;
}

// make sqr on each element of array
// and output it to out_array
for ( int i = 0; i < N; i++ ) {
    array_out[ i ] = sqr( array_in[ i ] );
}
```

That was a classic procedural approach

Functional Style Approach

- In functional style approach, we would use the `map` function

Functional Style Approach

- In functional style approach, we would use the `map` function
- `map` applies provided function on each element of an array (usually a list in FP) and outputs new array (list)

Functional Style Approach

- In functional style approach, we would use the `map` function
- `map` applies provided function on each element of an array (usually a list in FP) and outputs new array (list)

Something like:

```
array_out = map( fun, array_in )
```

Functional Style Approach

- In functional style approach, we would use the map function
- map applies provided function on each element of an array (usually a list in FP) and outputs new array (list)

Something like:

```
array_out = map( fun, array_in )
```

- Essentially, we would like to create an abstraction ovet the for loop.

C Meets (Almost) Functional Style Programming

- Let’s create a new type that is a pointer to a function with some arguments:

C Meets (Almost) Functional Style Programming

- Let's create a new type that is a pointer to a function with some arguments:

```c
typedef int (*map_i_func)( const int arg );
```

C Meets (Almost) Functional Style Programming

- Let's create a new type that is a pointer to a function with some arguments:

```
typedef int (*map_i_func)( const int arg );
```

- Function declaration fits nicely our `sqr` function:

```
int sqr( const int x ) {
    return x * x;
}
```

Inplace `map` - I

- `map` function that modifies its argument (array)

```
void map_i( map_i_func f, int * array, const int len );
```

Inplace `map` - II

- `map` function that modifies its argument (array)

```
void map_i( map_i_func f, int * array, const int len );

void map_i( map_i_func f, int * array, const int len ) {
    for ( int i = 0; i < len; i++ ) {
        array[ i ] = f( array[ i ] );
    }
}
```

Inplace `map` - III

```c
const int N = 5;
int array_in[ N ];

for ( int i = 0; i < N; i++ ) {
    array_in[ i ] = i;
}

map_i( sqr, array_in, N );

for ( int i = 0; i < N; i++ ) {
    printf( "%d ", array_in[ i ] );
}

// 0 1 4 9 16
```

Inplace `map` - IV

- It works, but not in a functional style
- `map` function modifies input array

`map` that Returns an Array

- Mapped function is the same:

```c
typedef int (*map_i_func)( const int arg );

int sqr( const int x ) {
    return x * x;
}
```

Returning `map` - I

```
int * map_i( map_i_func f, int * array, const int len );
```

Returning map - II

```
int * map_i( map_i_func f, int * array, const int len );

int * map_i( map_i_func f, int *array, const int len ) {
    int * arr_out = NULL;
    arr_out = (int *)malloc( len * sizeof( arr_out[ 0 ] ) )
    if ( !arr_out ) {
        fprintf( stderr, "No memory!\n" );
        exit( -1 );
    }

    for ( int i = 0; i < len; i++ ) {
        arr_out[ i ] = f( array[ i ] );
    }
    return arr_out;
}
```

Returning map - III

```c
const int N = 5;
int array_in[ N ];

for ( int i = 0; i < N; i++ ) {
    array_in[ i ] = i;
}

int * array_out = map_i( sqr, array_in, N );

for ( int i = 0; i < N; i++ ) {
    printf( "%d ", array_out[ i ] );
}

// 0 1 4 9 16
```

Inplace `map` - IV

- It works like in a functional style
- `map` function does not modify anything

Functional Comparison

- Since you have Functinal Programming course…
- Please, compare it to “real” functional style programming, as you were tought

reduce - I

- Mapped function is the same:

```
typedef float (*reduce_f_func)( float a, float b );

float sum_f( float x, float y ) {
    return x + y;
}
```

reduce - II

```
float reduce( reduce_f_func f, const float * array,
                                const int len );
```

reduce - III

```
float reduce( reduce_f_func f, const float * array,
                               const int len );

float reduce( reduce_f_func f, const float * array,
                               const int len )
{
    float tmp = 0.0f;
    for ( int i = 0; i < len; i++ ) {
        tmp = f( tmp, array[ i ] );
    }

    return tmp;
}
```

reduce - IV

```
const int N = 5;
float array_in[ N ];

for ( int i = 0; i < N; i++ ) {
    array_in[ i ] = (float)i;
}

float sum = reduce( sum_f, array_in, N );

printf( "%.2f ", sum );

// 10
```

Use Cases

- We can have much more fun with function pointers. . .

Use Cases

- We can have much more fun with function pointers. . .
- OpenGL callbacks

Use Cases

- We can have much more fun with function pointers. . .
- OpenGL callbacks
- GObject (Object Oriented Programming model in C)

Administrative

- Is programming hard?
- Can we teach you such skill?
- Can we teach you something more?

Farewell

Have a nice Xmas