Laboratorio de Programación

2016 -2

Juan Camilo Rada[1]

[1]Departamento de Electronica y ciencias de la computación

Octubre de 2016

Outline

Outline

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*
- Functions passed as an argument are widely known as *Callbacks*

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*
- Functions passed as an argument are widely known as *Callbacks*
- The function is expected to be called back at some convenient time

Function Pointers

```
int myFunction(double a, char b)
```

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` $\rightarrow$ `int myFunction`
  The pointer type has to be equal to the function return type

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` → `int myFunction`
  The pointer type has to be equal to the function return type
- `(double, char)` → `(double a, double b)`
  Types of the function members

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` $\rightarrow$ `int myFunction`
  The pointer type has to be equal to the function return type
- `(double, char)` $\rightarrow$ `(double a, double b)`
  Types of the function members
- `int (*pointer) (double, char) = myFunction`
  The pointer points to `myFunction`

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` $\rightarrow$ `int myFunction`
  The pointer type has to be equal to the function return type
- `(double, char)` $\rightarrow$ `(double a, double b)`
  Types of the function members
- `int (*pointer) (double, char) = myFunction`
  The pointer points to `myFunction`
- `void process(void (*funcp)(int), int a, int b)`
  A function which receives a function as a parameter

Function Pointers

Test the code

```
#include <stdio.h>

void callback(int value)
{
    printf("This Callback function prints the value = %i", value);
}
void process(void (*funcp)(int), int a, int b)
{
    int c = a + b;
    funcp(c);
}
int main()
{
    void (*functionPointer)(int);
    functionPointer = callback;
    process(functionPointer, 2, 3);
}
```

Function Pointers

Test the code

```
#include <stdio.h>
void callback()
{
    printf("The process has finished!!!\n");
}
void process(void (*funcp)(), int *a, int b)
{
    *a = (*a * *a) + b;
    funcp();
}
int main()
{
    int var = 5;
    void (*functionPointer)();
    functionPointer = callback;
    process(functionPointer, &var, 3);
    printf("a = %d", var);
}
```

Function Pointers

Test the code

```
#include <stdio.h>
int sumCallback(int a, int b)
{
    return a + b;
}
void process(int (*funcp)(int, int), int a, int b)
{
    printf("a = %d, b = %d, res = %d", a, b, funcp(a, b));
}
int main()
{
    int (*functionPointer)(int, int);
    functionPointer = sumCallback;
    process(functionPointer, 5, 3);
}
```

Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

The map function

- Is a *High-order* function

Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

The map function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments

Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

The `map` function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments
- The `map` function applies the input function over all the elements in an array

Function Pointers

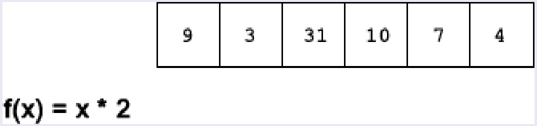
```
int* map(int (*fun)(int value), int *array, int size)
```

The `map` function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments
- The `map` function applies the input function over all the elements in an array
- *Keep in mind:* It works also for sequential *containers*

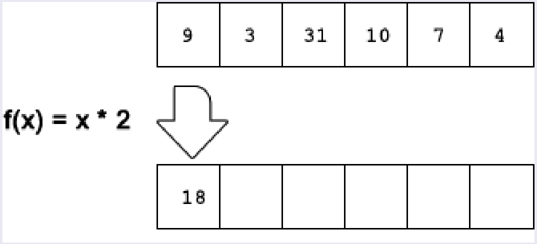
Function Pointers

The `map` function - How it works?



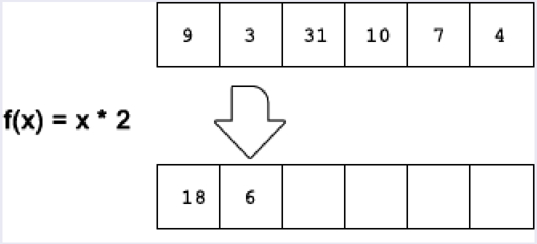
Function Pointers

The map function - How it works?



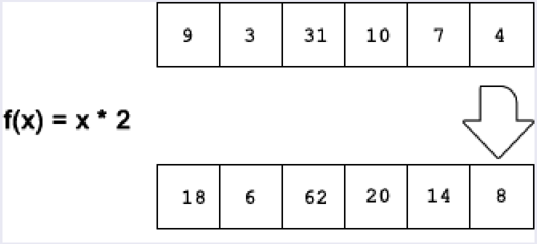
Function Pointers

The map function - How it works?



Function Pointers

The map function - How it works?



Function Pointers

The map function - Implementation

```
int* map(int (*funPtr)(int), int* ptr, int size)
{
    int* ptrRes = calloc(size, sizeof(int));
    for (int i=0; i<size; i++)
    {
        *(ptrRes+i) = funPtr(ptr[i]);
    }
    return ptrRes;
}
```

Function Pointers - Test the code with the map function

```
#include <stdio.h>
#include <stdlib.h>

int getDouble(int a)
{
    return a*2;
}

int main()
{
    int *a = calloc(2, sizeof(int));
    int (*fun)(int) = getDouble;
    a[0] = 5;
    a[1] = 9;

    int* array = map(fun, a,2);

    printf("%d, %d", array[0], array[1]);
}
```

Outline

Functions in structures

The Queue structure

```
struct queue
{
    Node * head;
    Node * tail;
};
```

Functions in structures

The Queue structure

```
struct queue
{
    Node * head;
    Node * tail;
};
```

The basic structure contains:

- A pointer to the head of the queue

Functions in structures

The Queue structure

```
struct queue
{
   Node * head;
   Node * tail;
};
```

The basic structure contains:

- A pointer to the head of the queue
- A pointer to the tail of the queue

Functions in structures

The Queue structure

```
struct queue
{
    Node * head;
    Node * tail;
};
```

- What if we want to *encapsulate* all the queue functions inside the structure?

Functions in structures

The Queue structure

```
struct queue
{
    Node * head;
    Node * tail;
};
```

- What if we want to *encapsulate* all the queue functions inside the structure?
- We can use pointers to functions

Functions in structures

The Queue structure

```
struct queue
{
    Node * head;
    Node * tail;

    void (*enqueue)(struct queue *, int);
    int (*dequeue)(struct queue *);
    void (*print)(struct queue *);
};
```

- `void (*enqueue)(struct queue *, int);`

Functions in structures

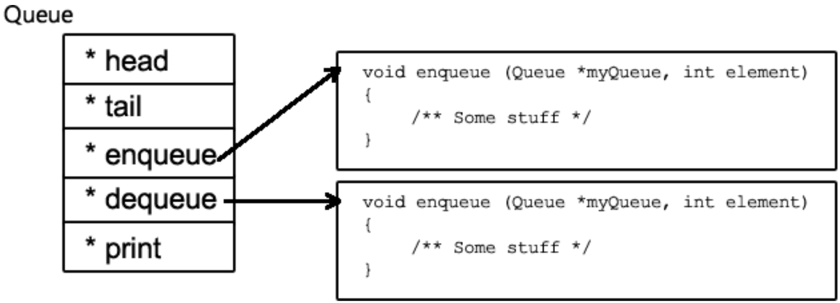
The Queue structure

```
struct queue
{
    Node * head;
    Node * tail;

    void (*enqueue)(struct queue *, int);
    int (*dequeue)(struct queue *);
    void (*print)(struct queue *);
};
```

- `void (*enqueue)(struct queue *, int);`
- We create three new pointers to every handled operation

Functions in structures



```
void enqueue (Queue *myQueue, int element)
{
    /** Some stuff */
}
```

```
void enqueue (Queue *myQueue, int element)
{
    /** Some stuff */
}
```

Functions in structures

The Queue structure - The Constructor function

```
Queue * constructor()
{
    Queue * myQueue = (Queue*)malloc(sizeof(Queue));
    myQueue->enqueue = enqueue;
    myQueue->dequeue = dequeue;
    myQueue->print = print;
    return myQueue;
}
```

- The constructor will provide us an instance of a Queue

Functions in structures

The Queue structure - The Constructor function

```
Queue * constructor()
{
    Queue * myQueue = (Queue*)malloc(sizeof(Queue));
    myQueue->enqueue = enqueue;
    myQueue->dequeue = dequeue;
    myQueue->print = print;
    return myQueue;
}
```

- The constructor will provide us an instance of a Queue
- We must associate every pointer to its respective function

Functions in structures

The Queue structure - The Constructor function

```
Queue * myQueue = constructor();
myQueue->enqueue(myQueue, 5);
```

Functions in structures

The Queue structure - The Constructor function

```
Queue * myQueue = constructor();
myQueue->enqueue(myQueue, 5);
```

- We have to call `constructor()` every time we need a new instance of Queue

Functions in structures

The Queue structure - The Constructor function

```
Queue * myQueue = constructor();
myQueue->enqueue(myQueue, 5);
```

- We have to call `constructor()` every time we need a new instance of Queue
- All the functions has to be called with the `->` operator

Functions in structures

The Queue structure - The Constructor function

```
Queue * myQueue = constructor();
myQueue->enqueue(myQueue, 5);
```

- We have to call `constructor()` every time we need a new instance of Queue
- All the functions has to be called with the `->` operator

Functions in structures

The Queue structure - The Constructor function

```
Queue * myQueue = constructor();
myQueue->enqueue(myQueue, 5);
```

Be careful!. The pointer to every function can be modified so, it can be unsafe!!!