EECS 280 Lab 03: Function Pointers

Due Friday, 31 January 2013, 11:55pm

In this lab, you will practice defining and using function pointers, in particular to implement higher order functions. Along the way, we'll also take a brief look at using the `typedef` keyword.

This lab covers material from these lectures:

- 04 Recursion and Iteration
- 05 Testing and Function Pointers
- 06 Arrays and Pointers

Overview

Requirements

You may work on this lab either individually or in groups of 2-3. Include your name(s) in the comments at the top of the file. Submit the files below on CTools. You do not need to turn in any other files. If you work in a group, each person must submit a copy in order to receive credit.

Files to submit:

- `lab03.cpp`

Completion Criteria:

To pass this lab you must complete task 1. Task 2 is recommended, but optional.
This checklist will give you an idea of what we look for when grading for completion:

- ✓ (Task 1) Implement `map`. Your implementation must be reasonably close to correct.

Task 0 - Preliminaries

The Files

We have provided a skeleton file in which you should write your code. You may copy it to your current directory using the command:

```
cp /afs/umich.edu/class/eecs280/lab/lab03/* .
```

It is also attached to the CTools assignment in case you are working locally.

Here's a brief summary of the files included in this lab. Files you need to turn in are shown with a red background.

`lab03.cpp`	Contains function stubs for the higher order functions `map` and `fold`. This file includes the `main` function and testing code.

Testing Code

`lab03.cpp` contains a `main` function with testing code we've written for you. Compile it with:

```
g++ -Wall -Werror -O1 -pedantic lab03.cpp -o lab03
```

The starter code should "work" out of the box, so make sure you are able to compile and run it. The code may be missing some pieces, contain some bugs, or crash when you run it, but you'll fix each throughout the course of the lab.

Function Pointer Review

Your GSI may take a few minutes at the beginning of the lab to review function pointers. You can find an example and more details in QR - Function Pointers.

Small Groups: Here are some more examples of type declarations. Try to decipher each of them. *Hint: These are not all necessarily function pointers!*

1. `int *(*var1)();`
2. `bool (*var2[3])(int, double);`
3. `double * var3;`
4. `double (*(*var4)())(int);`
5. `int **var5;`

Task 1 - Map

In this task, you will write a higher-order function called `map` that applies a function to each element in an array and places the result into a destination array. For example, if we have a function that triples a number:

```
int triple(int x){
  return 3 * x;
}
```

we can use map to triple each element in an array like this:

```
int arr[4] = {1, 2, 3, 4};
int tripledArr[4];
map(triple, arr, tripledArr, 4);

// tripledArr will now contain {3, 6, 9, 12}
// arr still contains {1, 2, 3, 4}
```

To keep things simple, we'll only work with arrays of `int`. So then the function prototype for `map` might look like below. (We also need to pass in the length of the array. Why?)

```
void map(int (*func)(int), int src[], int dst[], int length);
```

But we can clean this up a bit using a `typedef`. Conceptually, our `map` function wants to take in a function that makes some modification to an `int`, so let's call the type of such functions `intModifier` and rewrite the prototype of `map`.

```
typedef int (*intModifier)(int);

void map(intModifier func, int src[], int dst[], int length);
```

This is the form provided in `lab03.cpp`. To complete this task, you just need to fill in the implementation of the `map` function. The code we provide for you includes several `intModifier` functions (shown below) and testing code in `main` to verify your implementation is correct.

```
//***** intModifier functions *****//

//EFFECTS: returns 3*x
int triple(int x){
  return 3*x;
}
```

```
//EFFECTS: returns x+1
int addOne(int x){
  return x+1;
}

//EFFECTS: returns -1 if x < 7
//         returns 0 if x == 7
//         returns 1 if x > 7
int compareToSeven(int x){
  if (x < 7) {return -1;}
  else if (x == 7) {return 0;}
  else {return 1;}
}
```

Task 2 - Fold (Optional)

Your goal is to write another higher-order function called fold that "folds" up all the elements of an int array into a single result. More precisely, it takes as a parameter another function that takes two ints and combines them into one int result. First, the function is applied to combine the first two elements of the array. Then, the result of that is combined with the 3rd element of the array, and so on, until all elements have been put together. The final result is returned. This might sound kind of confusing, so let's look at an example.

Assume we have a function called `add`:

```
int add(int a, int b){
  return a + b;
}
```

Then if we create an array and fold it together, combining elements with `add`:

```
int arr[4] = {1, 2, 3, 4};
int result = fold(add, arr, 4);

//result now contains 10
```

Internally, `fold` first calls `add(1, 2)` to get `3`. Then it would call `add(3, 3)` to get `6`. Finally, it would call `add(6, 4)` to get `10`.

1. First, write an appropriate `typedef` for functions (like `add`) that combine integers. Find the line of code below in your `lab03.cpp` file, complete it by filling in the appropriate type

declaration, and uncomment it. Use `intCombiner` for the new type name;

```
// typdef ____________________;
```

2. Next, write the `fold` function. There's a very basic shell for this function in `lab03.cpp`. Make sure to uncomment the lines when you begin working on it. It looks like this:

```
/* int fold(intCombiner func, int src[], int length){
  TASK 2 - WRITE THE FUNCTION IMPLEMENTATION HERE
}*/
```

3. Finally, test your `fold` function. The `main` function we've provided you will test several arrays on each of the `intCombiner` functions (shown below) in `lab03.cpp`.

```
//***** intCombiner functions *****//

//EFFECTS: returns the sum of a and b
int add(int a, int b){
  return a + b;
}

//EFFECTS: returns the product of a and b
int mult(int a, int b){
  return a * b;
}

//EFFECTS: returns the number constructed by appending
//         the digits of b to those of a
int concat(int a, int b){
  //determine number of digits in b
  int digitsB = 1;
  while (b /= 10 > 0){ ++digitsB };
  return 10*digitsB*a + b;
}

//REQUIRES: a and b are positive (nonzero) integers
//EFFECTS: returns the greatest common divisor of a and b
int gcd(int a, int b){
  if (a == b){
    return a;
  }
  else if (a > b){
    return gcd(a-b, b);
```

```
  }
  else{
    return gcd(a, b-a);
  }
}
```