Lab 10 higher order functions in C++

- In C/C++ we can define pointers to functions, but we must specify the type/order of expected parameters, e.g. f below can point at functions that take two doubles as parameters and return a double

```
double (*f)(double,double);  // f is a function pointer for functions
f = pow; // make f point at function pow
double r = (*f)(3,5); // call pow through the pointer
```

- we can pass function pointers as parameters, even by reference, but the passed function must match the expected profile, e.g. the function passed as parameter fptr below must take expect two doubles as params

```
bool somefunc(double (*fptr)(double,double));
if (somefunc(pow)) { // call the higher order function, passing pow
```

Templated function pointers

- If we make our higher order function templated, using different types for each of the parameters and for the return value, then we could pass functions expecting different types of parameters

```
Template <class T1, class T2, class T3>

bool higherOrder(T1 (*fptr)(T2,T3)) { ... body of higherOrder... }

// sample calls

if (higherOrder(strcmp)) { // T1 an int, T2,T3 char*

if (higherOrder(pow)) { // T1, T2, T3 all doubles
```

Instantiation

- As usual with C++ templates, we have to be sure that, at compile time, the templates are filled in with appropriate types ... this can be challenging if using seperate compilation, since at the time we compile the file with the template we don't know what other .o's are using the template for
- We can either put the entire template in the .h file (so the files that include it know how to build/compile the template) or give specific instantiation commands with the template, identifying what we want built, e.g. build one expecting functions that take char,double and return int

```
template bool <int,char,double> higherOrder(int(*f)(char,double));
```

Lab 10 exercise

- you’re implementing a C++ version of reduce, that takes three parameters:
  - A templated function pointer that takes two parameters and returns a value, where all three are the same type
  - An array, also of the same type
  - An integer (the number of elements in the array)
- Reduce runs the passed function on the first two elements of the array, producing a result r, then runs the function on r and the next element of the array, getting a new result, and so on.

Example

- Suppose we pass a function and array to reduce

```
int subtract(int x, int y) { return (x – y); }
int data[4] = { 5, 1, 8, 3 };
int result = reduce(subtract, data, 4);
```

- Then reduce should perform
  - Subtract(5,1) => 4
  - Subtract(4,8) => -4
  - Subtract(-4,3) => -7
  - Returns -7 as the final result

Repo content

- README file with instructions (as usual)
- templateFuncPtrs.cpp with sample code
- lab10.cpp (your solution goes here)
- makefile