Pointers to functions and callback

We start programming with constants. Then we realize that this does not give us flexibility. We will have to change the constant each time in the program, recompile and then execute the program.

We introduce the concept of variables. We can run the same program a number of times and give different values for the variables.

We then realize that if the code being executed is exactly same but for the variable names, we could as well give a name for that piece of code and create a function. We get different results from a function call based on arguments passed to it.

Can a function do different operations based on the client needs? Can the client pass an operation as argument?

The client may pass a character standing for an operation. The function may use switch case statement and execute different pieces of code. This works fine as long as the client passes the character for which we have a valid switch case construct. What if the client passes a character for which we have no implementation?

Can we have a variable which stands for some executable code or some function? We can do this by providing a pointer to a function.
The client sends a pointer to a function with a particular signature. The function recalls it back. The client can pass any function with that signature. The function calling it back has no idea of what the function does. This is called the callback mechanism – which is heavily used in programming.

We have discussed this in Python – in functions like map, filter, reduce, sort.

Pointer to function:
Let us examine the declarations below.

Int *q(int, int);

q is a function which takes two int arguments and returns a pointer to int.

int (*p)(int, int);

p is a pointer to a function which takes two int as parameters and returns an int.

We can assign to p a function name as long as the function has the following signature.

int foo(int, int);

The function name acts like a pointer to a function.

Once p has been assigned as above, we can invoke the function foo through p as follows.

int x = p(3, 4);

The above statement is same as calling foo.

Callback:

Let us examine this function what.

```
int what(int x, int y, int (*op)(int, int))
{
        return op(x, y);
}
```

Observe the third argument op. It is a pointer to a function which takes two int arguments and returns an int.

The return statement in turn calls the function with two arguments. This function what does not know which function gets called. It depends on the value of op. The value of a variable is a runtime mechanism and compiler knows the type and not the value.

For some unusual reason, the following statements are equivalent in ‘C’.

p = add;

p = &add;

We shall use the former in our discussions.

Similarly, the following are equivalent in ‘C’.

p(3, 4);

(*p)(3, 4);

We shall use the former in our discussions.

The following code also shows that the function name can be directly passed as argument to the function with callback.

```
	printf("res : %d\n", what(3, 4, add));
	printf("res : %d\n", what(3, 4, mul));
```

```
#include <stdio.h>

int add(int x, int y)
{
	return x + y;
}

int mul(int x, int y)
{
	return x * y;
}

int what(int x, int y, int (*op)(int, int))
{
	return op(x, y);
}

int main()
{
	printf("result : %d\n", add(5, 6));
	int (*p)(int, int); // p is a  pointer to a function which takes two int as parameters
//	returns an int
#if 0
	p = &add;
	printf("result : %d\n", (*p)(5, 6));
	printf("p : %p add : %p\n", p, add);
#endif
	p = add;
	printf("result : %d\n", p(5, 6));
	printf("p : %p add : %p\n", p, add);

	printf("res : %d\n", what(3, 4, add));
	printf("res : %d\n", what(3, 4, mul));

}
```

Functional programming concepts in ‘C’:

mymap :
void mymap(int a[], int b[], int n, int (*op)(int));
This takes an input array a having n elements.
Applies the callback op on each element of a.
Stores the resulting elements in the corresponding position in output array b.

This is supplied as the callback function.
int sq(int x) { return x * x; }

int a[5] = {3, 1, 5, 2, 4};
int b[5];
mymap(a, b, 5, sq);
disp(b, 5);

The array b contains the elements { 9, 1, 25, 4, 16} in this case.

This is the implementation of mymap.

```
void mymap(int a[], int b[], int n, int (*op)(int))
{
	for(int i = 0; i < n; ++i)
	{
		b[i] = op(a[i]);
	}
}
```

myfilter:
This takes an input array a having n elements.
Applies the callback op on each element of a.
Stores the element of b in a if the callback returns a true value.

Note that this function should return the number of elements in b as it could be anywhere between 0 and n. We are changing an argument through the pointer for this purpose – ptr_m.

void myfilter(int a[], int b[], int n, int* ptr_m, int (*op)(int));

this is the function for callback – return true is the number is even.

```
int is_even(int x)
{
	return x % 2 == 0;
}
```

The function myfilter:

```
void myfilter(int a[], int b[], int n, int* ptr_m, int (*op)(int))
{
	int j = 0;
	for(int i = 0; i < n; ++i)
	{
		if(op(a[i]))
		{
```

```
                b[j++] = a[i];
            }
        }
        *ptr_m = j; // important step
}
```
This is the client code.
```
        int a[5] = {3, 1, 5, 2, 4};
        int b[5];
        int n = 5;
        int m;
        myfilter(a, b, n, &m, foo);
        disp(b, m);
```

In this case, the array b would contain {3, 1, 5}.

myreduce:

int myreduce(int a[], int n, int (*op)(int, int));
This function reduces an array a of n elements into a single value based on the callback. Observe that the callback takes two arguments.

The function for callback:
```
int add(int x, int y)
{
        return x + y;
}
```
The function myreduce:
Observe that the loop is executed n – 1 times. Observe that we assume that the zeroth element is the result. We combine the current result with the next element using the two argument call back function.

```
int myreduce(int a[], int n, int (*op)(int, int))
{
        // do n - 1 times
        int res = a[0];
        for(int i = 1; i < n ; ++i)
        {
                res = op(res, a[i]);
        }
        return res;
}
```
The client code:
```
        int a[5] = {3, 1, 5, 2, 4};
        int n = 5;
        printf("res : %d\n", myreduce(a, n, add));
```

Expected result is 15.

mysearch:
We look at versions of searching for an element in an array.
a) search for a matching element
b) search for a matching predicate.

a) search for a matching element

int mysearch(int a[], int n, int elem);
We search for elem in an array a of n elements from left to right.
We return the position of the leftmost match if found, or -1 if not found.
The function is:

```
int mysearch(int a[], int n, int elem)
{
	int pos = -1; int i = 0;
	while(pos == -1 && i < n)
	{
		if(a[i] == elem)
		{
			pos = i;
		}
		i++;
	}
	return pos;
}
```

We assume that the element is not yet found – we initialize pos to -1.
We loop as long as the element is not found and there is still a chance of finding the element. If the element matches, we update pos. When we exit the loop, pos will have the right value.

b) search with a predicate

```
int mysearch_predicate(int a[], int n, int (*op)(int))
{
	int pos = -1; int i = 0;
	while(pos == -1 && i < n)
	{
		if(op(a[i]))
		{
			pos = i;
		}
		i++;
	}
	return pos;
}
```

It is very similar to the last search function but for the predicate.
In the first one, we compared a[i] == elem. In the second, we call op(a[i]).
We exit when the predicate becomes true and return that corresponding index.
If predicate never becomes true, we exit when i becomes n and return -1.

```
#ifndef UTIL_H
#define UTIL_H
void mymap(int a[], int b[], int n, int (*op)(int));
void myfilter(int a[], int b[], int n, int* ptr_m, int (*op)(int));
int myreduce(int a[], int n, int (*op)(int, int));
```

```
// return an int
int mysearch(int a[], int n, int elem);
int mysearch_predicate(int a[], int n, int (*op)(int));
#endif


#include <stdio.h>
#include "2_util.h"
int add(int x, int y)
{
	return x + y;
}

int mul(int x, int y)
{
	return x * y;
}
void disp(int x[], int n)
{
	for(int i = 0; i < n; ++i)
	{
		printf("%d ", x[i]);
	}
	printf("\n");
}
int sq(int x) { return x * x; }
int foo(int x)
{
	return x % 2;
}

int is_even(int x)
{
	return x % 2 == 0;
}
int less_than_3(int x)
{
	return x < 3;
}
int main()
{
#if 0
	int a[5] = {3, 1, 5, 2, 4};
	int b[5];
	mymap(a, b, 5, sq);
	disp(b, 5);
#endif
#if 0
	int a[5] = {3, 1, 5, 2, 4};
	int b[5];
	int n = 5;
	int m;
```

```
	myfilter(a, b, n, &m, foo);
	disp(b, m);
#endif
#if 0
	int a[5] = {3, 1, 5, 2, 4};
	int n = 5;
	printf("res : %d\n", myreduce(a, n, add));
	printf("res : %d\n", myreduce(a, n, mul));
#endif
	int a[5] = {3, 1, 5, 2, 4};
	int n = 5;
	printf("res : %d\n", mysearch(a, n, 2));
	printf("res : %d\n", mysearch(a, n, 6));

	printf("res : %d\n", mysearch_predicate(a, n, is_even)); // pos 3: val 2
	printf("res : %d\n", mysearch_predicate(a, n, less_than_3)); // pos 1 : val 1



}




#include "2_util.h"

void mymap(int a[], int b[], int n, int (*op)(int))
{
	for(int i = 0; i < n; ++i)
	{
		b[i] = op(a[i]);
	}
}

void myfilter(int a[], int b[], int n, int* ptr_m, int (*op)(int))
{
	int j = 0;
	for(int i = 0; i < n; ++i)
	{
		if(op(a[i]))
		{
			b[j++] = a[i];
		}
	}
	*ptr_m = j;
}
int myreduce(int a[], int n, int (*op)(int, int))
{
	// do n - 1 times
	int res = a[0];
	for(int i = 1; i < n ; ++i)
	{
```

```
                res = op(res, a[i]);
        }
        return res;
}
// try:
//      pointer version
//      search from right
int mysearch(int a[], int n, int elem)
{
        int pos = -1; int i = 0;
        while(pos == -1 && i < n)
        {
                if(a[i] == elem)
                {
                        pos = i;
                }
                i++;
        }
        return pos;
}

int mysearch_predicate(int a[], int n, int (*op)(int))
{
        int pos = -1; int i = 0;
        while(pos == -1 && i < n)
        {
                if(op(a[i]))
                {
                        pos = i;
                }
                i++;
        }
        return pos;
}
```