Function Pointers

C also supports pointers to functions

This example takes a pointer to a `sum` function:

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

int main( void ) {
        int (*sum_ptr)(int, int) = &sum;
        return 0;
}
```

We've written the declaration of `sum_ptr` the same way we'd have written a prototype (function declaration) except we replaced the function name with `(*sum ptr)`

Let's simplify the syntax using `typedef`:

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

typedef int (*funcptr)(int, int);

int main( void ) {
        funcptr sum_ptr = &sum;
        return 0;
}
```

How do we use our function pointer:

```
funcptr sum_ptr = &sum;
int value = (*sum_ptr)(39, 73);
printf( "The sum of 39 and 73 is %d\n", value );
```

As a convenience, C99 allows us to pretend that the function pointer is the name of a normal function and write `int value = sum_ptr(39, 73);`

We can have multiple functions, and choose which one a function pointer points to, deciding at run-time, say via the command line arguments:

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

static int sum    ( int a, int b ) { return a + b; }
static int product( int a, int b ) { return a * b; }

typedef int (*funcptr)( int, int );

int main( int argc, char **argv ) {
  if( argc != 4 ) {
    fprintf( stderr, "Usage: calc a +|x b\n" ); exit(1);
  }
  int a = atoi(argv[1]); char *op = argv[2]; int b = atoi(argv[3]);

  funcptr fptr = NULL;
  if( strcmp( op, "+" ) == 0 ) {
    fptr = &sum;
  } else if( strcmp( op, "x" ) == 0 ) {
    fptr = &product;
  } else {
    fprintf( stderr, "calc: bad operator %s (should be +|x)\n", op ); exit(1);
  }
  int value = (*fptr)( a, b );
  printf( "%d %s %d is %d\n", a, op, b, value );
  return 0;
}
```

We can pass function pointers to other functions as well:

```
typedef int (*funcptr)( int, int );

int fold( funcptr f, char *op, int from, int to ) {
  int value = from;
  for( int i=from+1; i<=to; i++ ) {
    value = (*f)( value, i);
  }
  return value;
}

int main( int argc, char **argv ) {
  if( argc != 3 ) {
    fprintf( stderr, "Usage: fold +|x N\n" ); exit(1);
  }
  char *op = argv[1]; int n = atoi(argv[2]);

  funcptr fptr = NULL;
  if( strcmp( op, "+" ) == 0 ) {
    fptr = &sum;
  } else if( strcmp( op, "x" ) == 0 ) {
    fptr = &product;
  } else {
    fprintf( stderr, "fold: bad operator %s (should be +|x)\n", op ); exit(1);
  }
  printf( "folding %s over 1..%d: ", op, n );
  int value = fold( fptr, op, 1, n );
  printf( "result is %d\n", value );
  return 0;
}
```

What can you use Function Pointers for?

Function pointers allow you to make higher order functions

Function pointers also enable you to implement callbacks, much used in graphics programming and simulations

Here you embed a function pointer in a data structure, and the framework invokes that function under certain circumstances

For example, when a GUI button is clicked, a user-specified function can be invoked.

Finally, function pointers can help you write data driven programs

For example, a calculator might have a mapping from operator strings to operator functions stored in an array of pairs (see example `calc3.c` in week 2 lectures)

In this case, you can add a new operator simply by writing an operator function that implements it, and adding a new entry to the mapping. Everything else - validation of an operator, invocation of it’s corresponding operator function, and the error message produced when you give an unknown operator - is generated automatically from the mapping data. So you don’t have to make small pesky edits throughout the code