Function Pointer Basics

The actual instructions that your program executes are (of course), numbers, which are stored in the computer's memory, just like the program's data is. Consequently, each instruction has an address, just like each piece of data does. As these instructions have addresses, we can have pointers to them. It is not generally useful to have a pointer to an arbitrary instruction, but it can be quite useful to have a pointer to the first instruction in a function—which we typically just think of as a pointer to the function itself, and call a *function pointer*.

Technically speaking, the name of any function is a pointer to that function (that is *printf is* a pointer to the printf function), however, we do not typically think of them in this way. Instead, when we refer to a function pointer, we typically mean a variable or parameter that points at a function. However, the fact that a function's name is a pointer to it is useful to initialize such variables and/or parameters.

```c
void incAll(int * data, int n) {
  for (int i = 0; i < n; i++) {
    data[i] = data[i] + 1;
  }
}
```

(a) A function which increments all elements of an array.

```c
void squareAll(int * data, int n) {
  for (int i = 0; i < n; i++) {
    data[i] = data[i] * data[i];
  }
}
```

(b) A function which squares all elements of an array.

```c
void absAll(int * data, int n) {
  for (int i = 0; i < n; i++) {
    data[i] = abs(data[i]);
  }
}
```

(c) A function which takes the absolute value of all elements of an array.

```c
void doubleAll(int * data, int n) {
  for (int i = 0; i < n; i++) {
    data[i] = data[i] * 2;
  }
}
```

(d) A function which doubles all elements of an array.

The most useful application of function pointers arises from the ability to make a function pointer a parameter to a function we are writing (or that is provided by a library). To motivate this functionality, consider the four very similar pieces of code in the figure above. Each of these functions does *something* to every element of an array (of **int**s)—the only difference between them is *what* they do to each element.

Instead of duplicating the code—rewriting the entire function each time—it would be nicer if we could write one function which takes a parameter specifying "what to do to each item." Then, we could simply call that function with an appropriate function for each task. While avoiding this duplication of code may not seem so important here (the function is only a few lines long), this concern can become much more significant as you write more complex functions that operate over more complex data structures.

We can achieve this behavior by passing in a function pointer for the parameter that specifies "what to do to each item."

```c
void doToAll(int * data, int n, int (*f)(int)) {
  for (int i = 0; i < n; i++) {
    data[i] = f(data[i]);
  }
}
```

Most of the code in this example should seem quite familiar, except the somewhat odd looking parameter declaration:**int** (**f*) (**int**), which declares a parameter (called f), whose type is "a pointer to a function which takes an int as a parameter, and returns a *int*." Function pointer declarations are a bit unusual in that the name of the parameter (or variable—the declarations have the same syntax) is in the middle of the declaration. However, this syntax makes sense, as it looks a lot like the normal declaration of a function—the return type comes first, followed by the name, followed by the parameters in parenthesis. Here, however, we only need to specify the parameter types; we do not name them. Note that the parenthesis around **f* are important—without them the * becomes part of the return type (that is, the * is read as part of **int***), and the declaration appears to be describing a function that returns an **int***. There are times when *both* the parentheses and the * can be omitted (writing **int** *f* (**int)**), however, it is generally best to be consistent (and avoid trying to remember when this is permissible; we mention it in case you see it).

As with other types, we can use **typedef** with function pointers. The syntax is again more similar to function declarations than to other forms of **typedef**. We might re-write our previous example to use **typedef**, so that it is easier to read:

```
typedef int (*int_function_t) (int);


void doToAll(int * data, int n, int_function_t f) {
  for (int i = 0; i < n; i++) {
    data[i] = f(data[i]);
  }
}
```

Once we have this *doToAll* function defined, we can use it by passing in a pointer to any function of the appropriate type (*i.e.*, one that takes an int and returns an int). Since the name of a function is a pointer to it, we can just write the name of the function we want to use as the value to pass in for that argument:

```
int inc(int x) {
  return x + 1;
}

int square(int x) {

  return x * x;

}

…

doToAll(array1, n1, inc);

…

doToAll(array2, n2, square);

…
```

We will note that you may see such things written with the address-of operator, such as *doToAll* (*array1, n1, &inc*). This syntax is legal, but the *&* is superfluous, just as it is with the name of an array—the name of the function is already a pointer. Note that if we have a function pointer other than the name of a function (*i.e.*, a variable or a parameter), then we could take the address of that variable, giving us a pointer to a pointer to a function. It is best to use only the address-of operator in this latter case, which comes up rather infrequently.