CMSC 202 | **Functional Parameters** | Spring 1996
Sections 101, 102 & 103 | Handout #13

This handout describes a view of subprograms as objects, and details one particular application of this view—the passing of a subprogram as an argument to another subprogram.

1 The Object Metaphor

We think of data as objects. We talk about 'sending a parameter to a procedure,' 'getting a value back from a function,' and 'putting a number into an array.' We even use the term *data object*.

We are accustomed to thinking of subprograms (*i.e.* functions) as something other than data, since they directly control what is done when the program runs, where other data influence what happens only indirectly. Yet we can apply the object metaphor to subprograms as well as to data. That is, we can view subprograms as objects that might be manipulated in various ways, just as data objects are.

When we view a subprogram as an object, we can imagine doing a variety of things with it. The most obvious thing we can do with a subprogram is to invoke it. However, we might also imagine storing it in an array, or handing it as an argument to another subprogram, or modifying it in some way. In fact, virtually any type of operation that we can imagine performing on some other data type, we can imagine performing on a subprogram as well.

What could we do with a subprogram that was, say, stored in an array? The most obvious thing would be to select the subprogram from the array at some later time and invoke it. For example, suppose that we wanted to call one of ten different subprograms depending on the value of an integer variable `foo` that ranged from zero to nine. One way to do this would be to use a `switch` statement, using `foo` to control which case was selected:

```
switch(foo) {
  case 0: fn0(); break;
  case 1: fn1(); break;
    ⋮
  case 9: fn9(); break;
}
```

If we placed these ten functions in an array, then an alternative would be to directly invoke the subprogram stored at the array element indexed by `foo`:

```
arr[foo]();
```

Even more powerful than the idea of putting subprograms into a data structure is the idea of passing one subprogram to another as a parameter. Doing so effectively allows one to 'customize' the called subprogram at run-time (as opposed to modifying its code directly and recompiling it). We will use the term *functional parameter* to denote a subprogram that is passed as a parameter to another subprogram.

2 Example—Plotting a Function

As an example of the usefulness of functional parameters, suppose that we want to plot part of a function from floats to floats, using asterisks to represent discrete points along the function. We might first consider putting knowledge of the function to be plotted inside of the plotting routine. However, if we did so then we would need to write a new plotting routine for each function we wanted to plot. What this approach lacks is an *abstraction barrier* between the code that knows about plotting and the code that knows about calculating particular functions.

A second method we might try is to calculate the values to be plotted, place them in an array, and send this array to the plotting routine as an argument. This would give us a plotting routine that was independent of any particular function to be plotted. However, this method requires that we know in advance which points along the function are suitable for plotting. Thus, the abstraction barrier is still weak, because the routine that fills the array with points to be plotted needs to know something about what points it might be appropriate to plot. Furthermore, this method requires that we use an array to hold the data, where the first solution required only enough storage to hold one point at a time.

We can have the best of both worlds by passing the function to be plotted as an argument to the plotting routine. This makes the plotting routine independent of any particular function to be plotted, thereby solving the generality problem of our first solution. Furthermore, the plotting routine can invoke the function on any point it selects and immediately plot that point. This solves the control and storage problems present in our second solution.

The declaration of a functional parameter to a subprogram in C looks like a pointer to a function declaration. For example, here is a function plotting procedure that takes a function from floats to floats as its first argument:

```
void
plot(float (*fn_to_plot)(float),
     doc_string documentation,
     float lowest_x, float highest_x,
     float lowest_y, float highest_y)
{
  float x_value;
  float y_value;
  int row_to_plot;
  float x_increment_per_row;
  float y_increment_per_col;

  assert(lowest_x < highest_x);
  assert(lowest_y < highest_y);
  printf("Plot of %s for %0.2f <= x <= %0.2f and %0.2f <= y <= %0.2f\n\n",
         documentation, lowest_x, highest_x, lowest_y, highest_y);
  x_increment_per_row = (highest_x - lowest_x) / (NUM_AVAILABLE_ROWS - 1);
  y_increment_per_col = (highest_y - lowest_y) / (NUM_AVAILABLE_COLS - 1);
  for (row_to_plot = 0; row_to_plot < NUM_AVAILABLE_ROWS; row_to_plot++) {
    x_value = lowest_x + (row_to_plot * x_increment_per_row);
    y_value = fn_to_plot(x_value);
    write_star_in_column(round((y_value - lowest_y) / y_increment_per_col) + 1);
  }
  putchar('\n');
}
```

The first argument to `plot` is `float (*fn_to_plot)(float)`. This declaration says that if you dereference `fn_to_plot` and then hand it a float as an argument, the result will be a float. Note that no name is given to the argument of the functional parameter; only the type information (*i.e.* `float`) is given. Whenever you need to specify a type with no name in C, simply write the type definition as if it had a name (*e.g.* `float foo`), then delete the name. Sometimes this ends up looking strange (*e.g.* an array of pointers to `int`s will look like `int *[]`), but it is always correct. If you find such syntax ugly, you can always use `typedef` to give a name to the type.

A functional parameter is invoked simply by assuming that the formal parameter is an existing function, and calling that function. In `plot`, we invoke the function by saying `fn_to_plot(x_value)` and assigning the result to `y_value`. Note that technically, we ought to dereference `fn_to_plot` before calling it (as in `y_value = (*fn_to_plot)(x_value)`). However, as a shortcut, ANSI C allows us to omit the dereference if we like.

We can call a subprogram that takes a functional parameter by providing the name of the desired function as an argument. For example, suppose that we have the following three functions that we want to plot:

```
/* Functions to be plotted: */

#define PI 3.14159265358979
float
f1(float x)
{
  return(exp(-x) * sin(2 * PI * x));
}

float
f2(float x)
{
  return(sin(10 * x));
}

float
identity(float x)
{
  return(x);
}
```

We can plot each of these three functions with the following calls:

```
main()
{
  plot(f1, "f(x) = exp(-x)sin(2 pi x)", 0.0, 2.0, -0.5, 1.5);
  plot(f2, "f(x) = sin(10x)", 0.0, 2.0, -2.0, 2.0);
  plot(identity, "f(x) = x", -1.0, 1.0, -1.0, 1.0);
}
```

Notice that when we hand for example `f1` to `plot`, `f1` *is not called.* That is, we do not give `f1` an argument and invoke it, but rather simply provide its name to indicate that it is the function we want to plot. Here again, technically we ought to take the address of a function when handing it as a functional parameter to another subprogram (as in `plot(&f1, ...)`). However, ANSI C allows us to omit the `&` operator in this situation. If you use pre–ANSI C, you will need to use the `&` operator when passing a functional parameter.

Here is the result of the first call to `plot`:

```
Plot of f(x) = exp(-x)sin(2 pi x) for 0.00 <= x <= 2.00 and -0.50 <= y <= 1.50
                   *
                                *
                                          *
                                                *
                                                  *
                                               *
                                          *
                                  *
                          *
                 *
          *
    *
 *
 *
   *
      *
           *
                 *
                      *
                          *
                             *
                              *
                              *
                             *
                          *
                       *
                    *
                 *
               *
             *
            *
             *
              *
               *
                 *
                   *
```

The code presented in this handout, along with all other code needed to make this example work, is available in the file '~`mayfield/fn-params.c`'. Note that the program must be compiled with the math library:

```
% cc fn-params.c -lm
```

3 The Advantage of Functional Parameters

The most significant advantage of the ability to pass subprograms as arguments is that it allows us to build generalized routines that maintain strong abstraction barriers. In the function plotting example, we were able to write a procedure that can plot arbitrary functions from floats to floats, without knowing anything about particular functions it might plot. Furthermore, functions that are ultimately plotted can be written without ever imagining that they might be plotted. Thus, we have created a very general routine that exhibits a strong abstraction barrier between code devoted to plotting and code devoted to calculating particular functions.

As another example, consider the task of transforming an array of elements that are out of order into an array of elements that are in some particular order (such as numeric order, alphabetic order, *etc.*). In computer science, this task is called **sorting**.

Typical sorting algorithms work by comparing two elements of the array to see which one should come first, then using the result of the comparison to decide which of several actions to perform. The only component of such a sorting routine that actually looks at the data being sorted is the comparison routine. That means that one should be able to abstract the general notion of sorting away from the particular type of objects being sorted, by handing the comparison routine to the sorting routine as a parameter. Such a sorting routine never needs to know what kind of objects it is sorting! The `qsort` routine found in `<stdlib.h>` is an example of this approach to sorting. `Qsort` is described on page 329 of the text.

4 Using `typedef`

Just as with other complex types, C's `typedef` facility can be used to give a name to the type of a functional parameter. For example, the code:

```
typedef float (*floatfn)(float);
```

defines a new type called `floatfn` that is a pointer to a function that takes a `float` as its argument and returns a `float`. Given this definition, the header for our plot routine could be expressed as:

```
void
plot(floatfn fn_to_plot,
     doc_string documentation,
     float lowest_x, float highest_x,
     float lowest_y, float highest_y)
```

In addition to being easier to read, this use of `typedef` allows us to use a single name to refer to every instance of a function pointer of this type.

5 Functions as Structure and Array Elements

Let's return to our first example of function pointers, namely the idea of an array of function pointers. Using `typedef` gives us a simple way of declaring such arrays. For example, if we wanted an array of ten functions taking two integers as arguments and returning floats, we could use the following code:

```
typedef float (*my_fn_type)(int, int);
typedef my_fn_type my_array_type[10];
```

Arrays of type `my_array_type` can now be used to store references to functions just like an integer array can be used to store integers. If for example, we had a function called `foo` that took two integers as arguments and returned a float, we could allocate an array to hold such functions and place `foo` in its zeroth element with:

```
my_array_type my_array;
my_array[0] = foo;
```

Notice that just as when using functional parameters, we may omit the ampersand in front of `foo`, because C will automatically replace the use of the function name with a pointer to the function.

We can also place function pointers in structs. For example, if we want a program that reads commands from the user and executes different functions depending on the command given, we could use something like the following:

```
typedef void (*handler_fn)(char *);
typedef struct {
  char *command;
  handler_fn handler;
} command_mapping;

command_mapping map[] = {
  "show", handle_show,
  "take", handle_take,
  "drop", handle_drop,
  "quit", handle_quit
};
```

This code builds an array of structures called `map`. Each element of `map` contains the name of a command and the function to be executed when the user enters that command (it assumes that the functions `handle_show` `handle_take` `handle_drop` and `handle_quit` have already been defined, and that they are `void` functions that a string as argument). The program might repeatedly read in a user's command, separate the first word of the command from the remainder, look through `map` for a matching entry, and, if one is found, call the corresponding function (perhaps handing it the remainder of the command line as its argument). For example, entering the command "`take the cheesecake`" under this scheme would lead to an invocation of the function `handle_take` with argument `"the cheesecake"`.

6 Exercises

1. Write a function that takes a function and an integer as arguments, and returns the result of applying the function to the integer. The functional parameter should take one integer as its argument, and return an integer.

2. Write a function that takes a function and two integers as arguments, and returns the result of applying the function to the larger of the two integers. The functional parameter should take one integer as its argument, and return an integer.

3. Write a function that takes two functions and an integer as arguments, and returns the maximum of the values returned by the two functions when applied to the integer. The functional parameters should each take one integer as argument, and return an integer.

4. Write a program that calls each of the above functions on sample data.