Higher-Order Programming in C: Function Pointers and Void Pointers

Norman Ramsey

Fall 2008

Introduction

Hanson’s textbook presents intriguing functions like `Bit_map` and `Table_map`, but to my great disappointment, he doesn’t really motivate these functions or provide examples of their use. It’s too bad, because a function that takes a function as an argument is called a **higher-order function**, and it’s one of the most powerful programming techniques there is. Because implementing this technique in C requires function pointers and void pointers, I attempt to give you a few pointers here.¹

When you bring together a pointer to code and a pointer to data, you create a computational powerhouse that in functional languages is known as a **closure** and in object-oriented languages is known as an **object**. For C programmers, it is often useful to think of the data pointed to as being divided into two parts:

- **Context** needed for the function
- **Mutable data** needed to **accumulate** a result

If you have but a single `void *` pointer to work with, you must often create a structure containing both context and data.

Exercise: find available aligned register pairs

To help you learn these concepts, I’ve written an extremely simple exercise. Suppose you have a CII bit vector in which the vector represents a set of registers, a zero bit says that a register is available to hold a value, and a one bit says that a register is unavailable because it already holds a value. (This representation is a classic one used in many compilers.) The problem is as follows:

---

¹Triple pun! Pointers as in the language, pointers as in “tips,” and pointers as in “references to other people’s work.”
Use **Bit_map** to make a list of available aligned register pairs

An *aligned register pair* is a pair of registers $i$ and $i + 1$, where $i$ is even.

Recall the type of **Bit_map**:

```c
extern void Bit_map(T set,
                    void apply(int n, int bit, void *cl),
                    void *cl);
```

Your job is to create an *apply* function that will *accumulate* a list of available pairs. The *context* is the bit vector, and the *mutable state* is a list of register numbers $i$. So the `void *cl` passed to `Bit_map` and to `apply` should point to a structure with a type something like this:

```c
struct rp {
    Bit_T registers;
    List_T avail_pairs;
};
```

Here the `registers` field is the context and the `avail_pairs` field is the mutable data.

Here’s the skeleton of a function:

```c
List_T available_aligned_pairs(Bit_T regs)
{
    struct rp cl = { regs, NULL };
    Bit_map(regs, add_aligned_pair, &cl);
    return cl.avail_pairs;
}
```

We return the mutable data created by calls to `add_aligned_pair`. To finish the job, you’ll need to *define* `add_aligned_pair`, which will cast its third argument to a value of type `struct rp *`. It will then observe (without mutating) the `registers` field, but it may mutate the `avail_pairs` field by assigning it the result of calling `List_push`.

I recommend that you do this exercise and test it. You will not be asked to turn in the results for a grade, but any of the course staff will be happy to look at it.

**Further reading**

Rob Pike touches lightly on function pointers in his “Notes on Programming in C.” This five-page essay packs more good advice for page than anything else.
I’ve ever seen, which makes it required reading for all C programmers, including you.

If you’re confused about the basics of function pointers, you can find examples in Kernighan and Ritchie or in Hanson, as well as at sites like http://www.cprogramming.com/tutorial/function-pointers.html. Regrettably, none of these sources really explain the interplay of function pointers and data pointers (void *). You wind up stuck using only functions that don’t have any context and can’t mutate anything useful.

A sterling exception to this regrettable silence is the series of tutorial pages starting at http://theoryx5.uwinnipeg.ca/programming/node86.html. The examples are trivial, but at least they show the full generality of the programming model. And if you are uneasy about pointers in general, you can start with http://theoryx5.uwinnipeg.ca/programming/node80.html.