ECE2800J

Programming and Introductory Data Structures

Recursion; Function Pointers;

Learning Objectives:

Understand recursion and know how to write recursive functions

Understand how to write more general code with function pointers

Outline

- Recursion
- Function Pointers

Recursion

- Recursion is a nice way to solve problems
  - "Recursive" just means "refers to itself".
  - There is (at least) one "trivial" base or "stopping" case.
  - All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.
- Example: calculate factorial $n!$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

```
int factorial (int n) {
// REQUIRES: n >= 0
// EFFECTS:  computes n!
  if (n == 0) return 1; // base case
  else return n*factorial(n-1); // recursive step
}
```

Recursive Helper Function

- Sometimes it is easier to find a recursive solution to a problem if you change the original problem slightly, and then solve that problem using a **recursive helper function**.

```
soln()
{
   …
   soln_helper();
   …
}
```

```
soln_helper()
{
   …
   soln_helper();
   …
}
```

Outline

- Recursion
- Function Pointers

Function Pointers

Motivation

- If you were asked to write a function to add all the elements in a list, and another to multiply all the elements in a list, your functions would be almost exactly **the same**.

- Writing almost the exact same function twice is a bad idea! Why?
  1. It's wasteful of your time!!
  2. If you find a better way to implement some common parts, you have to change **many different** places; this is prone to error.

Our Example: list_t type

- A list can hold a sequence of zero or more integers.
- There is a recursive definition for the values that a list can take:
  - A valid list is:

either `an empty list`

or `an integer followed by another valid list`

Function Pointers

Background on lists

- Here are some examples of valid lists:

```
( 1 2 3 4 )   // a list of four elements
( 2 5 2 )     // a list of three elements
( )           // an empty list
```

- There are also several operations that can be applied to lists. We will use the following three:
  - list_first( ) takes a list, and returns the first element (an integer) from the list. **REQUIRES: non-empty list!**
  - list_rest( ) takes a list and returns the list comprising all but the first element. **REQUIRES: non-empty list!**
  - list_isEmpty( ) takes a list and returns the Boolean "true" if the argument is an empty list, and "false" otherwise.

Function Pointers

Using lists

- Suppose we want to write a **recursive** function to find the smallest element in a list.
  - The function requires the input list to be non-empty.

Question: how do you do it **recursively**?

- **Answer**:

```
smallest(list) =  the element (if list has
                  only a single element)
               or the minimum of the first
                  element and the smallest
                  element from the rest of
                  the list
```

Function Pointers

Using recursion to find the smallest element in a list

```
int smallest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS:  returns smallest element
  // in the list
{
   int first = list_first(list);
   list_t rest = list_rest(list);
   if(list_isEmpty(rest)) return first;
   int cand = smallest(rest);
   if(first <= cand) return first;
   return cand;
}
```

Function Pointers

Using lists

- Now suppose we want to write a recursive function to find the largest element in a list.
  - The function also requires the input list to be non-empty.

- Recursive definition:

```
largest(list) =  the element (if list has
                 only a single element)
              or the maximum of the first
                 element and the largest
                 element from the rest of
                 the list
```

Function Pointers

Using recursion to find the largest element in a list

```
int largest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS:  returns largest element
  // in the list
{
   int first = list_first(list);
   list_t rest = list_rest(list);
   if(list_isEmpty(rest)) return first;
   int cand = largest(rest);
   if(first >= cand) return first;
   return cand;
}
```

Function Pointers

More Motivation

- `largest` is almost identical to the definition of `smallest`.
- Unsurprisingly, the solution is almost identical, too.
- In fact, the **only** differences between smallest and largest are:
  1. The names of the function
  2. The comment in the EFFECTS list
  3. The polarity of the comparison: <= vs. >=
- It is silly to write almost the same function twice!

Function pointers to rescue!

Function Pointers

A first look

- So far, we've only defined functions as entities that can be called. However, functions can also be referred to by **variables**, and passed as **arguments** to functions.

- Suppose there are two functions we want to pick between: min() and max(). They are defined as follows:

```
int min(int a, int b);
  // EFFECTS: returns the smaller of a and b.
int max(int a, int b);
  // EFFECTS: returns the larger of a and b.
```

Function Pointers

A first look

```
int min(int a, int b);
  // EFFECTS: returns the smaller of a and b.
int max(int a, int b);
  // EFFECTS: returns the larger of a and b.
```

- These two functions have precisely the same type signature:
  - They both take two integers, and return an integer.
- Of course, they do completely different things:
  - One returns a min and one returns a max.
  - **However, from a syntactic point of view, you call either of them the same way.**

Function Pointers

The basic format

- How do you define a **variable** that points to a function that takes two integers, and returns an integer?

- Here's how:

```
int  (*foo)(int, int);
```

- You read this from "inside out". In other words:

`foo`	"foo"
`(*foo)`	"is a pointer"
`(*foo)(          );`	"to a function"
`(*foo)(int, int);`	"that takes two integers"
`int  (*foo)(int, int);`	"and returns an integer"

Function Pointers

The basic format

```
int  (*foo)(int, int);
```

- Once we've declared foo, we can **assign** any function to it:

```
foo = min;
```

- Furthermore, after assigning min to foo, we can just call it as follows:

```
foo(3, 5)
```

- …and we'll get back 3!

Function Pointers v.s. Variable Pointers

- For function pointers, the compiler allows us to **ignore** the "address-of" and "dereference" operators.

```
int  (*foo)(int, int);
foo = min; // min() is predefined
foo(5,3);
```

We don't write:

```
foo = &min;
(*foo)(5, 3);
```

- In contrast, for variable pointers:

```
int  foo;
int *bar;
bar = &foo;
*bar = 2;
```

Function Pointers

Re-write `smallest` in terms of function pointers

```
int compare_help(list_t list, int (*fn)(int, int))
{
    int first = list_first(list);
    list_t rest = list_rest(list);
    if(list_isEmpty(rest)) return first;
    int cand = compare_help(rest, fn);
    return fn(first, cand);
}
int smallest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS: returns smallest element in list
{
  return compare_help(list, min);
}
```

```
int min(int a, int b);
    // EFFECTS: returns the
    // smaller of a and b.
```

Function Pointers

Re-write `largest` in terms of function pointers

```
int compare_help(list_t list, int (*fn)(int, int))
{
    int first = list_first(list);
    list_t rest = list_rest(list);
    if(list_isEmpty(rest)) return first;
    int cand = compare_help(rest, fn);
    return fn(first, cand);
}
int largest(list_t list)
  // REQUIRES: list is not empty
  // EFFECTS: returns largest element in list
{
  return compare_help(list, max);
}
```

```
int max(int a, int b);
    // EFFECTS: returns the
    // larger of a and b.
```

Exercise

- Given an array of integers, return the sum.
- Given an array of integers, return the product.
- Use function pointer to avoid writing the "same" code twice.

Reference

- Recursion
  - Problem Solving with C++, 8th Edition, Chapter 14

- Function pointers
  - C++ Primer (4th Edision), Chapter 7.9