Advanced pointer topics

(Reek, Ch. 13)
Pointers to functions

- Declaration:
  returnType (*varName)(parameterTypes);

- Examples:
  int (*f)(int, float);
  int (**g[])(int, float);
  int *(*g[])(int, float);

  - Pointer to a function that takes an integer argument and a float argument and returns an integer
  - Pointer to a function that takes an integer argument and a float argument and returns a pointer to an integer
  - An array of pointers to functions – Each function takes an integer argument and a float argument and returns a pointer to an integer
**Pointers to functions: WHY?**

- They allow for a certain amount of **polymorphism**:
  - “poly” (many) + “morph” (shape)
  - A polymorphic language can handle a range of different data types (“shapes”?) with a single statement
- This is common in OO languages like C++, Java:

```c
Animal myPet;
...
myPet.makeSound();
```

This method call will result in different sounds, depending on whether `myPet` holds a `Cow` object, an `Elephant` object, etc.
Example: searching a singly-linked list

typedef struct IntNode {
    int value;
    struct IntNode *next;
} INTNODE;

INTNODE *search_list(INTNODE *node, int const key) {
    while (!node) {
        if (node->value == key) break;
        node = node->next;
    }
    return node;
}
A more abstract notion of “node”

typedef struct Node {
    void *value;   struct Node *next;
} NODE;

void construct_node(NODE *node, void *value, NODE *next) {
    node->value = value;    node->next = next;
}

NODE *new_node(void *value, NODE *next) {
    NODE *node = (NODE *)malloc(sizeof(NODE));
    construct_node(node, value, next);
    return node;
}
A more abstract notion of “search list”

- What is it that makes the old `search_list` only work for integers?
  - The `key` parameter is of type `int`
  - The `==` operator is used to compare `int` values – but `==` will not work for many types (e.g. structs, strings)
- A solution: pass in an additional argument – a comparison function!
  - Programmer must supply a comparison function that’s appropriate for the data type being stored in the nodes
  - This function argument is called a **callback function**:
    - Caller passes in a pointer to a function
    - Callee then “calls back” to the caller-supplied function
Abstract “search list” with callback function

```c
NODE *search_list(NODE *node, void const *key,
    int (*compare)(void const *, void const *)) {

    while (node) {
        if (!compare(node->value, key)) break;
        node = node->next;
    }

    return node;
```

Assumption: `compare` returns zero if its parameter values are equal; nonzero otherwise
Using callback functions

- If our nodes hold strings, we have a compare function already defined: `strcmp` or `strncmpy`

```c
#include <string.h>
...
match = search_list(root, "key", &strcmp);
```

Note: you may get a warning, since `strcmp` is not strictly of the right type: its parameters are of type `char *` rather than `void *`
Using callback functions

- If our nodes hold other kinds of data, we may need to “roll our own” compare function

```c
int compare_ints(void const *a, void const *b) {
    const int ia = *(int *)a, ib = *(int *)b;
    return ia != ib;
}

...  
match = search_list(root, key, &compare_ints);
```
Jump tables

- In some cases, a nice alternative to long, repetitive switch statements, like this:

```c
double add(double, double);
double sub(double, double);
double mul(double, double);
double div(double, double);

switch(oper) {
  case ADD:    result = add(op1, op2); break;
  case SUB:    result = sub(op1, op2); break;
  case MUL:    result = mul(op1, op2); break;
  case DIV:    result = div(op1, op2); break;
}
```
Jump tables

Jump table alternative:

double add(double, double);
double sub(double, double);
double mul(double, double);
double div(double, double);

double (*oper_func[])(double, double) = {
    add, sub, mul, div
};

result = oper_func[oper](op1, op2);
Pointers to functions: safety concerns

- What if uninitialized function pointer value is accessed?
  - Safest outcome: memory error, and program is terminated
  - But what if the “garbage” value is a valid address?
    - Worst case: address contains program instruction – execution continues, with random results
    - Hard to trace the cause of the erroneous behavior
Command line arguments

- C programs can be called from the command line, with certain arguments entered along with the program name:
  - e.g. Registration program register
  - You may register with an existing ID by the `-i` option: register -i ID
  - Otherwise, an ID will be generated
Command line arguments

- The `main` function can be declared with two arguments:
  
  ```
  int main(int argc, char **argv)
  ```

- `argc` holds the number of arguments

- `argv` is an array of strings: the $n$th command line string is stored at `argv[n-1]`
Implementation of registration program

```c
int main(int argc, char **argv) {
    char id[ID_LIMIT];
    switch(argc) {
    case 1:    generate_ID(id);
                break;
    case 3:    if (strcmp(argv[1], "-i")
                exit(INVALID_ARG);
                strcpy(id, argv[2]);
                break;
    default:   exit(INVALID_ARG_COUNT);
    }
    register(id);
}
```
References

- The Function Pointer Tutorials. 
  http://www.newty.de/fpt/index.html