Lab 2: Refactoring

This lab will test your ability to "refactor" a program into a more useful tool. This is a common task for system programmers: start with a piece of code that works in a specific case and generalize it.

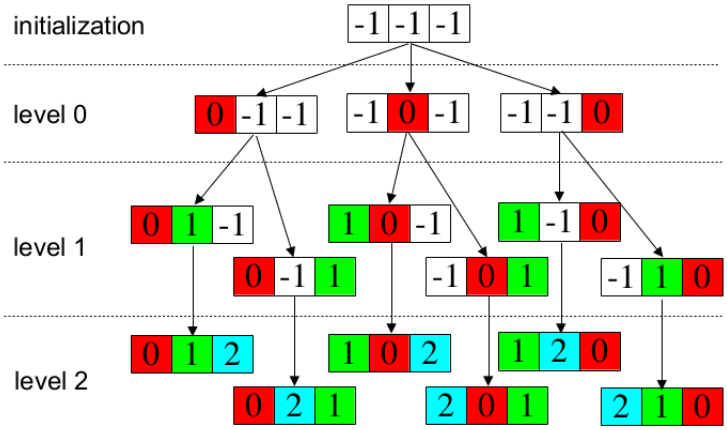
All code files in this handout are available in this directory:

http://www.tricity.wsu.edu/~bobl/cpts360/lab02_permutations

The file `writeup.html` in that directory is the HTML version of this handout.

The `perm` Program

We will start with a working program that generates permutations: the set of all possible orderings of *n* objects. The instructor will go over this algorithm at the start of class using this diagram.



This is the code for its implementation, `perm.c`:

```
#include <stdio.h>

int level;
enum {
    N_ELEM = 3,
    NOT_DONE = -1
};
int val[N_ELEM];


void recur(int k)
{
    int i;

    val[k] = level;
    level++;
    if (level == N_ELEM) {
        for (i = 0; i < N_ELEM; i++)
            printf("%d ", val[i]);
        printf("\n");
    }
```

```
    for (i = 0; i < N_ELEM; i++)
        if (val[i] == NOT_DONE)
            recur(i);
    level--;
    val[k] = NOT_DONE;
}

int main(int argc, char *argv[])
{
    int i;

    level = 0;
    for (i = 0; i < N_ELEM; i++)
        val[i] = NOT_DONE;
    for (i = 0; i < N_ELEM; i++)
        recur(i);
    return 0;
}
```

It will generate all 6 (= 3!) possible numerical permutations of integers from 0 to 2 inclusive, one per line. What you will do is turn this code into a function that generates all possible permutations of any number array elements.

Step 1: Create a Makefile for `perm`

Download and extract the `lab_refactoring.tgz` tarball from the course web page. Then, copy `Makefile_tplt` to `Makefile` and follow the comments to adapt it to compile and link `perm.c` into `perm` (as the default target).

Step 2: Refactor `perm` into `permute`

Copy and "refactor" `perm.c` to create `permute`, a program that will generate all possible permutations of an arbitrary number of command line arguments, one permutation per line. For example:

```
$ permute red green blue
```

will produce:

```
red green blue
red blue green
green red blue
blue red green
green blue red
blue green red
```

(although not necessarily in this order).

Code for `permute` consists of three files: `permute.c`, `gen_perms.h`, and `gen_perms.c`. Moving the ability of generating permutations from `perm.c` into the latter two files makes the code usable in the future whenever you need to generate permutations.

`permute.c` is:

```
#include <stdio.h>
#include <stdlib.h>

#include "gen_perms.h"

/* printPermutation -- print a permutation of an array of char *'s */
static void printPermutation(
    int indices[],
    int nIndices,
    void *userArg)
{
    int i;
    char **syms = userArg;

    for (i = 0; i < nIndices; i++)
        printf("%s ", syms[indices[i]]);
    printf("\n");
}

int main(int argc, char *argv[])
{
    genPerms(argc-1, &printPermutation, &argv[1]);
    return 0;
}
```

`gen_perms.h` is:

```
extern void genPerms(int nElems,
                     void (*handlePerm)(int elems[],
                                        int nElems,
                                        void *userArg),
                     void *userArg);
```

You are not allowed to modify either of these files.

Use `perm.c` as a template to create `genPerms()`, which belongs in `gen_perms.c`. In this function,

- `nElems`: is the length of the permutation
- `handlePerm()`: is a "callback": it is called once for each permutation with these arguments:
  - `elems`: is an array of permuted indices.
  - `nElems`: is the length of the permutation (and is equal to the `nElems` argument of `permute()`)
  - `userArg`: is identical to the `userArg` passed to `permute()` (see below)
- `userArg`: is an arbitrary pointer passed on by `permute()`. This is a common technique in C system programming. Take a look at `permute.c` and you'll see how using it avoids the need for global variables. (This means there is no problem using `genPerms()` with threads.)

For example, the call:

```
genPerms(2, tryit, arg);
```

where `tryit` (whose prototype matches that of `handlePerm`) will generate two calls. The first is `tryit(p, 2, arg)` where the contents of p are `{0, 1}` and the second is `tryit(p, 2, arg)` where the contents of p are `{1, 0}`.

The order of the permutations (i.e., the lines) is not important. Do not use permutation code from any other source: You must base your code on `perm.c`. You may wish to put additional "helper" functions in `gen_perms.c`.

Incorporate support to compile `permute` into the same `makefile` you created for `perm` so that `make`, by default, compiles and links both targets: `perm` and `permute`.

Submission

Put all source files and the `Makefile` in a tarball and submit it via Canvas.

Suggestion

Copy the code that sets up `val[]` from `perm()`'s `main()` to `genPerms()`, call a modified `recur()` (once) from there, and let that `recur()` continue to recursively call itself.